Stream Processing for Fast and Efficient Rotated Haar-like Features using Rotated Integral Images

Christopher Messom

Institute of Information and Mathematical Sciences, Massey University,Albany Highway, Auckland New Zealand
E-mail: c.h.messom@massey.ac.nz

Andre Barczak

Institute of Information and Mathematical Sciences, Massey University, Albany Highway, Auckland New Zealand,
E-mail: a.l.barczak@massey.ac.nz

Abstract: An extended set of Haar-like features for image sensors beyond the standard vertically and horizontally aligned Haar-like features and the 45^{o} twisted Haar-like features are introduced. The extended rotated Haar-like features are based on the standard Haar-like features that have been rotated based on whole integer pixel based rotations. These rotated feature values can also be calculated using rotated integral images which means that they can be fast and efficiently calculated with just 8 operations irrespective of the feature size. The integral image calculations can be offloaded to the graphical processing unit (GPU) using the stream processing paradigm. The integral image calculation on the GPU is seen to be faster than the traditional central processing unit (CPU) implementation of the algorithm, for large image sizes, allowing more complex clasifiers to be implemented in real-time.

Keywords: Haar-like features, Integral Images, Stream Processing, General Purpose GPU processing.

1 Introduction

Image sensors have become more significant in the information age with the advent of commodity multi-media capture devices such as digital cameras, webcams and camera phones. The data from these media sources (whether they are still images or video) is reaching the stage where manual processing and archiving is becoming impossible. It is now possible to process these images and videos for some applications in near real-time, using motion detection and face tracking for security systems for example. However there are still many challenges including the ability to recognise and track objects at arbitrary rotations (Lozano and Otsuka, 2008).

Haar-like features have been used successfully in image sensors for face tracking and classification problems (Lai et al., 2001; Jones and Viola, 2003; Barreto et al., 2004; Huang and Lai, 2004), however other problems such as hand tracking (Barczak et al., 2005; Micilotta and Bowden, 2004; Kölsch and Turk, 2004) have not been so successful. The main reason for this is the fact that Haar-like features are not invariant over rotation. This means that any object that rotates and is sensitive to angle changes (such as hands) will be difficult to solve using standard Haar-like features. The features that define faces tend to be insensitive to small angle variations and Haar-like features have been used to detect head rotations of as much as 15^{o} from the vertical (Jones and Viola, 2003). When people are standing their head is naturally aligned vertically with respect to gravity and so this rotational sensitivity tends not to be a significant problem for faces. Other body parts such as hands, arms and legs are not normally alligned with the horizontal or vertical axes so are difficult to model with traditional Haar-like features. Researchers have tended to use edge detection or colour based tracking of these parts (Messom et al., 2007).

Figure 1 Standard Haar-like features

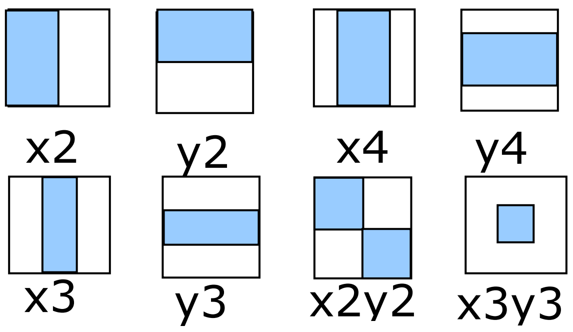
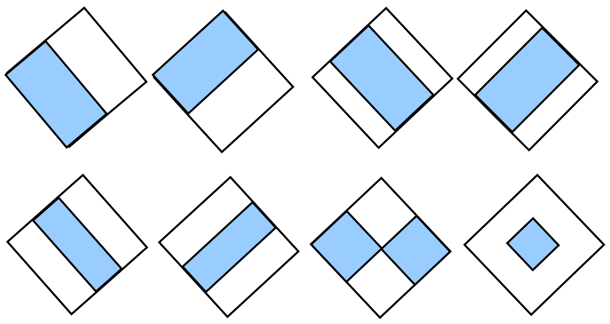


Figure 2 45^{o} twisted Haar-like features



Several researchers have studied the impact of in plane rotations for image sensors with the use of twisted Haar-like feature (45^{o}) (Lienhart and Maydt, 2002; Lienhart et al., 2003a; 2003b) or diagonal features (Viola and Jones, 2001b) fairly good performance has been achieved. These techniques will have little benefit for problems that are sensitive to rotations, such as hand identification (Barczak et al., 2005; Kölsch and Turk, 2004; Antón-Canalís et al., 2005; Stenger et al, 2004; Wachs et al., 2005) which are not aligned to fixed angles (0^{o}, 45^{o}, 90^{o} etc).

Real time image processing is starting to be feasible on commodity hardware however high frame rates for high resolution image sensors (greater than 640x480) are still a challenge. Haar-like feature based classifiers like the Jones and Viola, (2003), (Viola and Jones, 2001a and 2001b) face detector work in almost real time using the integral image (or summed area table) data structure that allows features to be calculated at any scale with only 8 operations. However standard Haar-like features are strongly aligned to the vertical/ horizontal (Jones and Viola, 2003)(fig 1) or 45^{o} diagonal (Lienhart and Maydt, 2002, Lienhart et al., 2003a and 2003b) (fig 2) and so are most suited to classifying objects that are strongly aligned as well, such as faces, buildings etc.

Rotated Haar-like features have to be calculated on a “rotated” integral image, meaning that a set of parallel classifiers that identify objects at different rotations require multiple integral images (Messom and Barczak, 2008). The calculation of these integral images have become one of the computationally significant parts of the classifier algorithms. Parallel hardware such as cluster computers (Messom and Barczak, 2008) have been used to implement parallel classifiers in real-time, however these systems suffer from being large and require a suitable high bandwidth low latency network between the nodes.

Recent GPUs from ATI and NVIDIA have been designed so that the computational elements can be programmed via programmable shaders. This means that arbitrary code (with some restrictions) can be offloaded to the GPU rather than the CPU. The main performance issue is that the code should be parallel and able to exploit a large number of processors, the current generation of GPUs being able to process 800 floating point operations simultaneously. At the machine level the GPU and CPU do not share the same instruction set so a higher level interface is provided (Tardi et al., 2006, Yamagiwa and Sousa, 2007, Gordon et al., 2006, Kuo et al., 2005). NVIDIA provides CUDA, while ATI provides Brook+ (Buck et al., 2004) as the C/C++ programming interface to their respective GPU cards. Additional standard lower level interfaces such as OpenGL and DirectX are also available while ATI also has a proprietary interface called CAL.

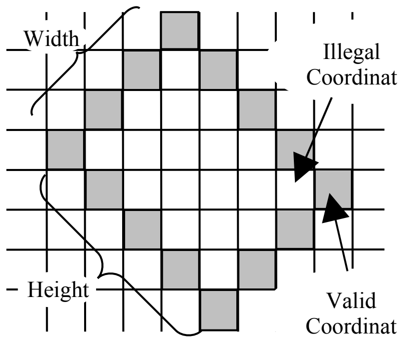
The latest GPUs from ATI (HD4850, HD4870 and HD4870X2) have more than 1 Tflop (2 Tflop for HD4870X2) of theoretical single precision floating point performance compared to a modern quad core cpu that has about 60Gflops of single precision performance, so significant speedup can be obtained for algorithms with appropriate characteristics. Algorithms that have already been implemented on the GPU include protein folding (Elsen et al., 2006), particle filter based face tracking (Lozano and Otsuka, 2008), sequence alignment (Schatz et al., 2007), forensic analysis (Marziale et al., 2007), histogram generation (Scheuermann et al., 2007) and various vector and matrix operations (Buck et al., 2004, Liao et al., 2006, Fan et al., 2004, Govindaraju and Manocha, 2007).

2 Standard Haar-Like Features

Standard Haar-like features consist of a class of local features that are calculated by subtracting the sum of a subregion of the feature from the sum of the remaining region of the feature. This is illustrated by figure 1. These features are characterised by the fact that they are easy to calculate and with the use of an integral image, very efficient to calculate. Lienhart and Maydt (2002) introduced an extended set of twisted Haar-like feature, illustrated in figure 2. These are the standard Haar-like features that have been twisted by 45^o.

Figure 3 Opposite Corners of 45^o twisted Haar-like feature on identical diagonal

These twisted Haar-like features can also be fast and efficiently calculated using an



integral image that has been twisted 45^o. The only implementation issue is that the twisted features must be rounded to integer values so that they are aligned with pixel boundaries. This process is similar to the rounding used when scaling a Haar-like feature for larger or smaller windows, however one difference is that for a 45^o twisted feature, the integer number of pixels used for the height and width of the feature mean that the diagonal coordinates of the pixel will be always on the same diagonal set of pixels, see figure 3. This means that the number of different sized 45^o twisted features available is significantly reduced as compared to the standard vertically and horizontally aligned features.

Integral images or summed area tables (Crow, 1984) are a data structure that contains the sum of all the pixels above and to the left of the current pixel. The time complexity of the algorithm is 2MN (where M and N are the height and width of the image) since each pixel in the integral image requires two addition operations (see eqn 1).

$$II(i, j)= II(i-1, j)+ II(i, j-1)-II(i-1,j-1)+ I(i, j) \tag{1}$$

where I(i, j) is the pixel value at position (i, j), II(i, j) is the integral image value at position (i, j).

Integral images are important as they allow the sum of a rectangular area of pixels of any size to be calculated with only 4 look ups in the Integral image data structure:

$$\Sigma^{b}_{i=a},\Sigma^{d}_{i=c}\, I(i, j)=II(a, c)- II(a, d)- II(b, c)+ II(b, d) \tag{2}$$

where I(i, j) is the pixel value at position (i, j), II(i, j) is the integral image value at position (i, j), (a, c) is the coordinate of the top left pixel and (b, d) is the coordinate of the bottom right pixel of the rectangular region that is being summed.

Standard Haar-like features consist of a class of local features that are calculated by subtracting the sum of a subregion of the feature from the sum of the remaining region of the feature.

3 Integer Rotated Haar-LikeFeatures

General rotations of Haar-like features can not be easily implemented efficiently, therefore we define a restricted set of rotations called integer rotations that can be easily and efficiently implemented. An integer rotated Haar-like feature is a feature that has been rotated by an angle arctan(A/B) where A and B are integers. This means that an integer rotated line consists of all angles that have a rational tangent. A 45^o rotated Haar-like feature is a special case of a feature which has been 1-1 integer rotated. A unit-integer rotated Haar-like feature is a feature that has been rotated by an angle arctan(A/B) where A and B are integers and either A or B is 1. A 45^o rotated Haar-like feature is a special case of a feature which has been 1-1 unit-integer rotated. Figure 4 illustrates a set of 1-2 rotated Haar-like features.

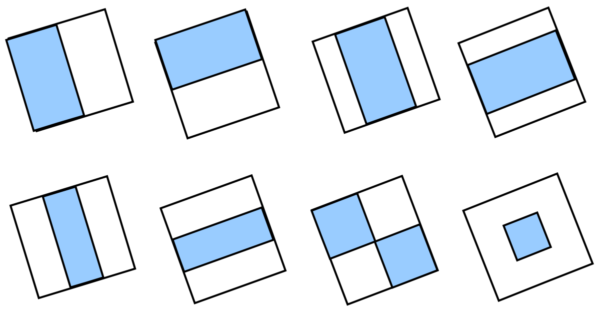


Figure 4 1-2 Rotated (26.57°) Haar-like features

This paper will discuss unit-integer rotated features, which restricts the angles available to those listed in table 1. The table shows that a large number of unit-integer rotations are available near the horizontal or vertical while only a few are available near 45°. In practise depending on the coverage required, a selection of these unit-integer rotations will be chosen, for example if rotations of about 10° to 20° degree increments are needed for a particular problem then 1-1, 1-2, 2-1, 1-4 and 4-1 will be used as well as the standard horizontal and vertically aligned features giving 0°, 14°, 26.5°, 45°, 63.5°, 76°, 90°. The rotations in the other three quadrants are given by simple reflections in the x and y axes. If a higher precision and accuracy are required, rotations of less than 10° would need a non unit-integer rotations such as 2-3 and 3-2 rotations giving angles of 37° and 53°.

The availability of rotated features means that a fully trained classifier using the standard features can be transformed to a rotated version. For example a face tracking system that is reliable for vertically aligned faces within a range of ±20° will be rotated so that it can classify faces aligned within ±20° of any of the unit-integer rotated axes such as 45° ±20°, 26.5° ±20° etc, effectively producing a parallel classifier (similar to that of Rowley (1998) that can cover all possible rotations.)

Table 1 Unit-Integer rotation angles

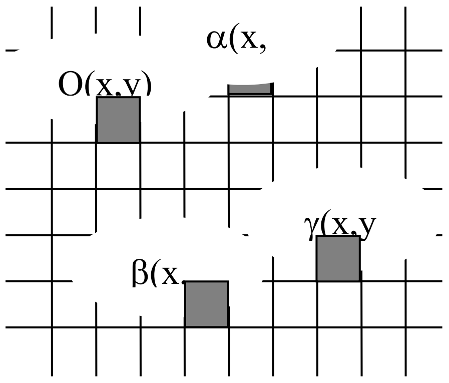
N-1	Angle	1-N	Angle
1-1	45°	1-1	45°
2-1	63.43°	1-2	26.57°
3-1	71.57°	1-3	18.43°
4-1	75.96°	1-4	14.04°
5-1	78.69°	1-5	11.31°
6-1	80.54°	1-6	9.46°
7-1	81.87°	1-7	8.13°
8-1	82.87°	1-8	7.13°
9-1	83.66°	1-9	6.34°
10-1	84.29°	1-10	5.71°
etc	..	etc	..

4 Implementation Issues for Integer Rotate Haar-Like Features

When a feature is rotated the position of the top left corner of the feature will be defined by the rotation angle and the position of the feature in the kernel. The height and width of the feature then determines which pixels form the feature itself. Rounding will cause the height and width of the feature to be aligned with the pixel boundaries. Since the pixelation of raster lines are not unique we need to choose a standard rasterisation so that the features provide consistent values. In this paper we rasterise based on the position of the starting point of the feature in the image.

A standard rasterisation is chosen so that a change in horizontal pixel position occurs every n vertical pixels for 1-n rotated vertical lines, while a change in vertical pixel position occurs every n horizontal pixels for 1-n rotated horizontal lines. This is calculated using equations 3 and illustrated in figure 5.

Figure 5 Calculation of feature coordinates in Integral Image



$$\alpha(x,y) = O(x,y)+[w*\cos\theta, -(w*\cos\theta)/n-((w*\cos\theta)/n+x)\bmod n]$$

$$\beta(x,y) = O(x,y)+[(h*\sin\theta)/n + ((h*\sin\theta)/n+y) \bmod n, h*\sin\theta]$$

$$\gamma(x,y) = H(\beta(x,y)) \cap V(\alpha(x,y)) \qquad (3)$$

where O(x,y) are the coordinates of the start of the feature, α(x,y), β(x,y) and γ(x,y) are the calculated coordinates of the top right, bottom left and bottom right of the feature based on the height h and width w of the feature, θ is the 1-n unit-integer rotation angle, H(ε) and V(ε), represents the "horizontal" and "vertical" lines through ε in the unit-integer rotated image, ∩ represents the intersection operator of two lines, * is integer multiplication and mod is the integer modulo operator.

Figure 6 A 1-2 rotated feature of height 5 and width 4 starting from a different positions

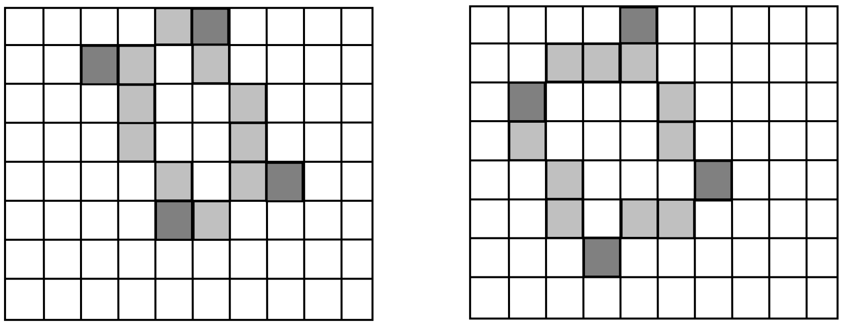
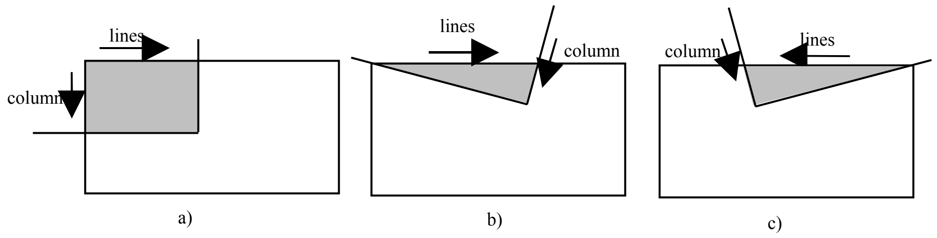


Figure 6 shows two features of height 5 and width 4 that has been 1-2 rotated. The figure also shows that a general unit-integer rotation (other than 1-1 rotations) in a digital image result in raster lines that are pixelated. The pixelation between two points depends on the position in which the line begins as illustrated by the second example in figure 6. Large feature sizes as compared to the integer rotation size (n for 1-n and n-1 integer rotations) will result in similar raster lines, but will still provide different feature values when evaluated as they consist of different pixels. Using a standard rasterization a single integral image can be used to evaluate the feature, whatever the size and starting position of the feature. For some values of h and w and and angle θ the lines (β(x,y)) and V(α(x,y)) will not interect. To overcome this problem, h, w or both h and w may need to be modified (±1 pixel), so that the do intersect. This is the similar problem to the 45^o twisted features where h and w must be modified so that opposite corners are on a valid coordinate (see figure 3).

4.1 Rotated Integral Image

The rotated integral image for a given integer rotation is calculated by summing the pixels in the relevant aligned quadrant above the given pixel (see figure 7). For the normal integral image with no rotation this is the quadrant above and to the left of the current pixel (figure 7a). (Lienhart and Maydt, 2002) extended this algorithm to the twisted integral image (which is equivalent to the 45^o rotated integral image, or the 1-1 integer rotated image).

Figure 7 Buffering of lines for a) normal integral image, b) >45^o rotated integral image c) <45^o rotated integral image



The algorithm to calculate the integer rotated integral images is also efficiently implemented. For simplicity, rotated integral images that are greater than 45^o are processed left to right while rotated integral image that are less than 45^o are processed right to left. This is done because the only quadrant that depends only on the pixels above the current point in the integral image is the "top right" quadrant for angles less that 45^o (figure 7c) and the "top left" quadrant for angles greater that 45^o (figure 7c).

4.2 Scaling correction factors

When the Haar features are scaled for use in a classifier the feature values must be corrected based on distortions due to uneven scaling of the different areas of the features. This is achieved by using an identity integral image to count the number of pixels in each feature area and scaling the areas appropriately. The corrected feature values are given by equation 4.

$$F = [A_\alpha * f_\beta / A_\beta] - f_\alpha \quad (4)$$

where A_α and A_β are the areas of the feature and the sub-features respectively (obtained from a rotated identity integral image) and f_α and f_β are the values of the sum of the pixels in the feature and sub-features respectively (obtained from a rotated integral image).

Table 2 was compiled from running the normal, twisted and rotated integral image algorithm on an opteron 250 processor (2.4GHz with 1MB cache). It can be seen from table 2 (shaded columns) that over a range of image sizes the integer-rotated integral images are slower to calculate than the normal integral images by an order of magnitude (between 8.5 and 14 times slower), but they are not significantly slower to calculate than the twisted integral image. These results show that the integer-rotated integral image can be used in real time image processing systems if only one classifier is implemented. For multiple classifiers for various different angle offsets are required then a multiprocessor or GPU implementation must be adopted.

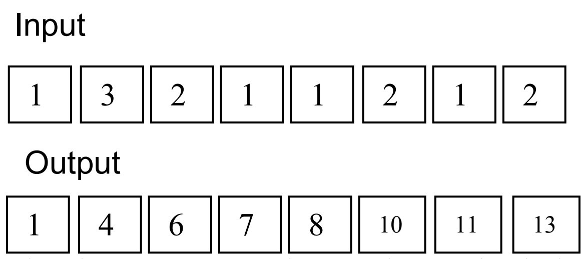
Table 2 Time complexity of Integral Image calculation including ratio of twisted and rotated features versus normal features

Image Size	Normal/ ms	Twisted/ ms	Twisted/ Normal	Rotated / ms	Rotated/ Normal
320x240	0.3	4.2	14	4.2	14
640x480	1.6	4.4	2.75	17.2	10.75
1280 x 960	8	22	2.75	68	8.5

5 Stream Processing

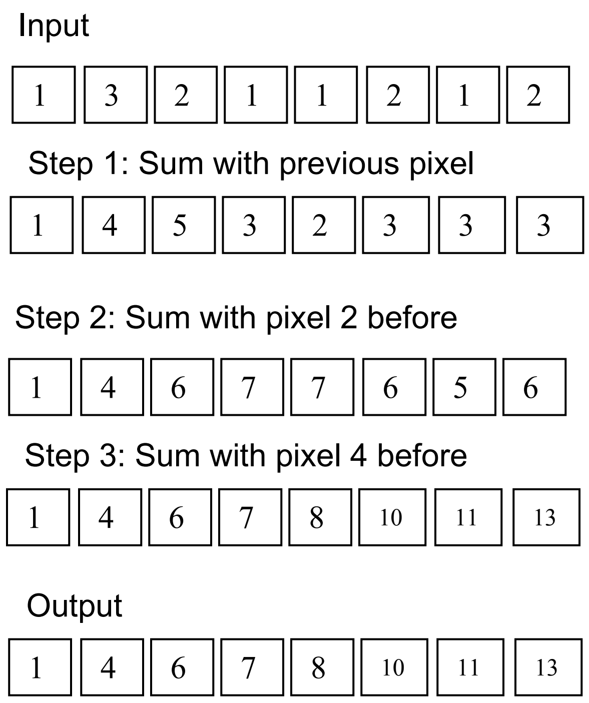
Stream processing is a programming model for highly parallel systems which can have hundreds of processors and thousands of threads in flight. Modern GPUs have adopted this programming model and it has been implemented via the Brook language extensions to C/C++ (Buck et al., 2004). The key concept of stream processing is that a stream needs to be created and a kernel implemented that acts on each stream element independently, so that processing can occur in parallel. For an image processing system the image is a 2 dimensional stream and the kernel must operate on each pixel independently. Due to the limitations of the GPU hardware, the kernels can only access stream inputs and outputs and other constant parameters. This does restrict the type of code that may benefit from stream computing, but massively parallel algorithms such as image processing can benefit greatly.

Figure 8 Prefix sum applied to a give input list



Efficient execution in stream processing requires each pixel to be processed independently, so that processing can occur in parallel. This means that the traditional serial algorithm for integral images is not suitable. However a similar algorithm, the prefix sum has been extensively studied on massively parallel architectures. The prefix sum algorithm calculates a running sum of the values in a list of data so that the result is a list that includes the sum of all values in the input list up to the current position in the list (see fig 8 for an example).

Figure 9 Stream Processing Prefix sum applied to a give input list, required log2N steps where N is the length of the list


Input
1
3
2
1
1
2
1
2
Step 1: Sum with previous pixel
1
4
5
3
2
3
3
3
Step 2: Sum with pixel 2 before
1
4
6
7
7
6
5
6
Step 3: Sum with pixel 4 before
1
4
6
7
8
10
11
13
Output
1
4
6
7
8
10
11
13

5.1 Stream Processing Prefix Sum Algorithm

The standard stream processing prefix sum algorithm follows a logarithmic number of passes through the data summing the current data with its prior neighbour on the first pass, then on the second pass sums the current data with the data a distance 2 before, then on the third pass sums the current data with the data a distance 4 before etc. (see Figure 9 for an illustration) until there are no more passes left to process. It is important to note that these operations occur for all elements of the stream simultaneously (or effectively simultaneously).

This algorithm requires a total of $\log_2 N$ passes where there are N data items to process. However each pass requires N operations as each data item in the stream is processed. This algorithm if run serially is very inefficient as a total of $N \log_2 N$ operations are required while the traditional serial prefix sum requires only N operations. However the stream processing algorithm can operate on P simultaneous processors which means that the total number of cycles is $(N \log_2 N)/P$, where it is assumed that $P<N$, that is the number of processing elements is less that the number of items in the stream. For large P the value $\log_2 N$ is significantly smaller than P so there is a large speedup associated with the algorithm.

5.2 Prefix Sum Kernel

The prefix sum kernel is very simple using a single kernel call for each step of the algorithm (see fig 10). At each step the current pixel value is added with a pixel from a given offset. The kernel must be called with the relevant horizontal or vertical offset ((-1,0), (-2,0), (-4,0) etc for the horizontal prefix sum, (0,-1), (0,-2), (0, -4) etc for the vertical prefix sum. The two dimensional offset is required as the streams for image processing applications are two dimensional. This is the default for GPUs that have a two dimensional frame buffer. The prefix sum kernel is inefficient in that each stream element that is processed is a single floating point value which can result in multiple memory reads as the floating point values are read into the kernel.

Figure 10 Kernel for the prefix sum algorithm
```
kernel void prefixsum_scan(
float input[][], out float output<>, float2 twoDoffset)
{
 float2 i = indexof(output);
 float2 index;
 index=i+twoDoffset;
 if (index.x < 0.0f || index.y < 0.0f)
 {
  output = input[i];
 }
 else
 {
  output = input[i] + input[index];
 }
}
```

Figure 11 Calculating the Integral Image with a horizontal prefix sum followed by a vertical prefix sum

Initial Image

2	1	3	1	1	2
3	2	1	1	3	4
1	2	3	2	1	2
4	1	3	1	1	3

Step 1: Horizontal Prefix Sum

2	3	6	7	8	10
3	5	6	7	10	14
1	3	6	8	11	13
4	5	8	9	10	13

Step 2: Vertical Prefix Sum, produces

2	3	6	7	8	10
5	8	12	14	18	24
6	11	18	22	29	37
10	16	26	31	39	50

Modern GPUs have a native float vector type called float4 that can load 4 floating point values in a single memory access. This data type will improve the memory access performance at the expense of additional code complexity. When each kernel element is a float4 type, it essentially contains 4 pixels and so when the prefix_sum kernel needs to access the neighbouring pixel, it doesn't access the neighbouring stream element, but the neighbouring pixel in the float4 data (except for the first element in the float4 data).

The ATI GPU additionally can output 8 streams simultaneously. What this means is that the kernel can be modified to process an image that has been split into 8 different streams. This additional processing that takes place in the kernel increases the computational intensity of the kernel, again improving the potential efficiency of the implementation. The prefix_sum kernel that uses the float4 data type and 8 simultaneous output streams has been implemented and is available online.

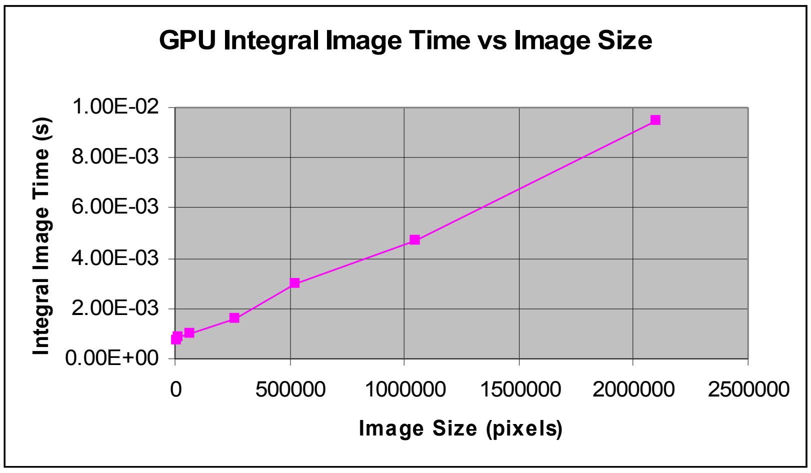
Table 3 Performance of the Integral Image stream calculation natively on the GPU

Image Shape	64x64	128x128	256x256	512x512	1024x1024	1024x2048
Image Size (pixels)	**4096**	**16384**	**65536**	**262144**	**1048576**	**2097152**
Time, 1 cycle (s)	**0.0259**	**0.0262**	**0.0268**	**0.0299**	**0.0445**	**0.0646**
Time, 2 cycles (s)	**0.0268**	**0.0272**	**0.028**	**0.0314**	**0.0484**	**0.0735**
Time, 10 cycles (s)	**0.0327**	**0.0342**	**0.0359**	**0.0445**	**0.087**	**0.1498**
Time, average cycle (not including initial cycle) (s)	**7.6E-04**	**8.9E-04**	**1.0E-03**	**1.6E-03**	**4.7E-03**	**9.5E-03**
Average cycle/ Image Size (s/pixels)	**1.8E-07**	**5.4E-08**	**1.5E-08**	**6.2E-09**	**4.5E-09**	**4.5E-09**

5.3 Stream Processing Integral Image Algorithm

The stream processing Integral Image algorithm uses the same kernel as the prefix sum. However the prefix sum kernel is used to calculate the prefix sum of all pixels in the horizontal direction followed by the prefix sum in the vertical direction (see fig 11 for an example). This means that the prefix sum kernel must be called $\log_2 N$ times for the horizontal prefix sum and $\log_2 M$ times for the vertical prefix sum resulting in a time complexity of $MN(\log_2 N + \log_2 M)/P$ which is significantly less than 2MN the time complexity for the serial integral image algorithm when P is large, that is $P>(\log_2 N + \log_2 M)/2$.

Figure 12 Integral Image Stream Processing time versus Image Size on the GPU



The rotated integral image (say for the 45° rotated features) can also be calculated on the GPU. Rather than the horizontal and vertical offsets of (-1,0), (-2,0), (-4,0) etc and (0,-1), (0,-2), (0, -4) etc, the two diagonal offsets are passed to the prefix_sum kernel. These are (-1,-1), (-2,-2), (-4,-4) etc and (1,-1), (2,-2), (-4, -4) etc. These offsets result in two sets of integral images that have been calculated since neighbouring diagonal lines do not intersect (this is similar to the black and white squares on the chess or checker board). In addition a final pass that adds the integral image from the pixel above (offset (0,-1)) must be applied so that the total integral image (not just for black or white squares) are calculated. Integer rotated integral images can also be similarly calculated using the prefix_sum kernel.

6 Performance Results

The integral image algorithm was tested using two different programming models and hardware platform. It was tested on the GPU (ATI HD4850) using the stream processing model, tested on the CPU (2.66GHz Intel quad core processor) using a simulated stream processing model, and tested on the CPU using a best available serial algorithm.

Figure 13 Integral Image Stream Processing time versus Image Size on the CPU

6.1 Stream Processing Integral Image on GPU

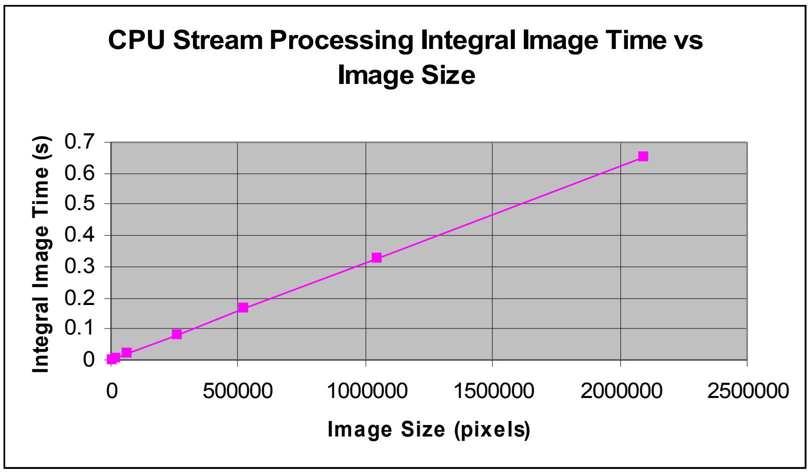


Table 3 shows the performance of the stream processing integral image algorithm running on an ATI HD4850 using the Radeon 8.5 driver. The results in table 1 show that the initialization of the GPU with the kernel requires a significant amount of time. This means that the first pass through the kernel takes a large amount of time. Subsequent passes, as shown by the timing when run twice and ten times shows that each individual pass of the stream programming integral image algorithm is relatively efficient. As the size of the image to process increases the execution time increases (fig 12) but the efficiency of the algorithm improves. The efficiency improves as the size of the image increases, as can be seen by comparing the average time to process each pixel in the last line of the table 3.

6.2 Stream Processing Integral Image simulated on CPU

Table 4 shows the performance of the simulated stream processing integral image algorithm running on the CPU. This executes code that simulates running on a GPU having the same number of function calls, except the code is running on a single processing core of the CPU. The performance is poor in comparison to the algorithm running on dedicated GPU hardware, but this is also a reflection of the fact that the algorithm is running the inefficient stream processing algorithm with a time complexity of $MN(\log_2 N + \log_2 M)/P$ where P is 1.

Table 4 Performance of the Integral Image stream calculation on the CPU

Image Shape (MxN)	64x64	128x128	256x256	512x512	1024x1024	1024x2048
$\log_2 M+\log_2 N$	**12**	**14**	**16**	**18**	**20**	**21**
Image Size (pixels)	**4096**	**16384**	**65536**	**262144**	**1048576**	**2097152**
Time, float data type, 10 cycle (s)	**0.819**	**3.832**	**17.47**	**79.44**	**354.76**	**744.50**
Time, float4 data type, 10 cycles (s)	**0.318**	**1.487**	**6.81**	**30.71**	**136.7**	**387.37**
Time, 8 streams, float4 data, type 10 cycles s)	**0.1552**	**0.709**	**3.243**	**14.82**	**65.42**	**137.44**
Time, average cycle(s)	**0.01552**	**0.0709**	**0.3243**	**1.482**	**6.542**	**13.744**
Serial Time, average cycle/ $\log_2 M+\log_2 N$ (s)	**0.00129**	**0.005**	**0.020**	**0.082**	**0.327**	**0.654**
Average cycle/ Image Size (s/pixels)	**3.8E-06**	**4.3E-06**	**4.9E-06**	**5.6E-06**	**6.2E-06**	**6.5E-06**
Average serial cycle/ Image Size (s/pixels)	**3.2E-07**	**3.1E-07**	**3.1E-07**	**3.1E-07**	**3.1E-07**	**3.1E-07**

Figure 13 shows that the algorithm scales linearly as the number of pixels processed increases and this is reflected in the average time to process 1 pixel which is almost constant over the different image size range (Table 4). The results (table 4) also show that the float4 stream implementation is more efficient that the float implementation, while the float4 implementation with 8 output streams is the most efficient. This is due to the reduced number of "threads" or function calls in this serial simulation resulting in a higher computational intensity of these implementations.

6.3 Serial Integral Image on CPU

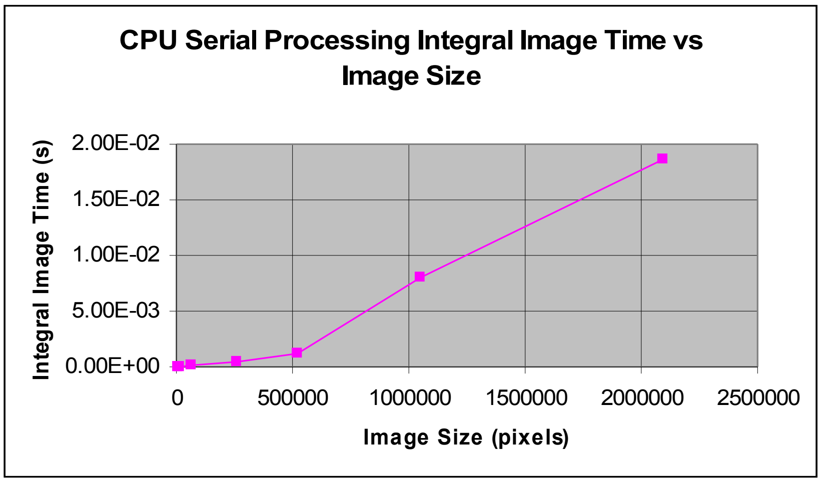
Table 5 shows the performance of the serial integral image algorithm running on the CPU. This executes a best available serial integral image algorithm running on the CPU. This code has time complexity of 2MN and does not include the overhead of multiple function calls to the kernel and so is significantly better performing that the simulated stream processing integral image algorithm as presented in table 4. Figure 14 shows that as the image size increases there is not a linear improvement in performance and this is illustrated by the average time to process a pixel (table 5). This is due to the cache effects of the CPU, as the problem size exceeds the caches the time to process each pixel increases.

Overall the time to process each pixel is very good across the range of input image sizes. For large image sizes the CPU implementation is slower than the GPU implementation, however it was using a single processor core. Efficient use of multiple cores will have the CPU performing better, however even in these scenarios offloading the CPU from the integral image calculations so that the CPU can continue other processing may be a viable option.

Figure 14 Integral Image Serial Processing time versus Image Size on the CPU

Table 5 Performance of the Integral Image calculation on the CPU

Image Shape	64x64	128x128	256x256	512x512	1024x1024	1024x2048
Image Size (pixels)	**4096**	**16384**	**65536**	**262144**	**1048576**	**2097152**
Time, 10 cycles (s)	**0.00007**	**0.0003**	**0.001**	**0.005**	**0.080**	**0.186**
Time, average cycle(s)	**7.0E-06**	**2.8E-05**	**1.1E-04**	**5.0E-04**	**8.1E-03**	**1.9E-02**
Average cycle/ Image Size (s/pixels)	**1.7E-09**	**1.7E-09**	**1.6E-09**	**1.9E-09**	**7.7E-09**	**8.9E-09**



7 Conclusions

This paper has presented novel rotated Haar-like features for image sensors that can be used to produce rotationally invariant classifiers. The unit-integer rotated integral image can be calculated at a speed similar to twisted features on the CPU or the GPU, with similar constant time calculation of feature values, independent of feature size.

This paper has shown that the execution time of the integral image algorithm for large input images from high resolution image sensors can be significantly improved by using a stream programming implementation running on the GPU. With the improvements of hardware and the addition of more stream processors in future commodity hardware, this technique represents a method to achieve real time performance while maintaining a reduced power budget as compared to modern CPUs.

Using the GPU for a full classifier (for example a face detector) requires additional processing. The integral image calculation represents just the first stage of the classifier. The cascade classifier of Jones and Viola, 2003 works by rejecting regions at each stage of the classifier that are not the object of interest. If a region passes all the stages then it represents an object of interest (for example a face). The cascade classifier algorithm has multiple code paths depending on the image in the region being processed and so is expected not to scale well on the GPU, however the computational intensity of the

Table 5 Performance of the Integral Image calculation on the CPU

Image Shape	64x64	128x128	256x256	512x512	1024x1024	1024x2048
Image Size (pixels)	**4096**	**16384**	**65536**	**262144**	**1048576**	**2097152**
Time, 10 cycles (s)	**0.00007**	**0.0003**	**0.001**	**0.005**	**0.080**	**0.186**
Time, average cycle(s)	**7.0E-06**	**2.8E-05**	**1.1E-04**	**5.0E-04**	**8.1E-03**	**1.9E-02**
Average cycle/ Image Size (s/pixels)	**1.7E-09**	**1.7E-09**	**1.6E-09**	**1.9E-09**	**7.7E-09**	**8.9E-09**

cascade classifier is relatively high. This may mean that offloading the classifier to the GPU, though not very efficient may be worth while since it will free the CPU for other tasks.

Alternative classifier approaches such as moment invariant approaches (Barczak et al., 2007) also can make use of integral images and have the additional advantage that they have limited code paths through the classifiers. We are currently investigating this approach and are optimistic about the potential performance benefits.

Acknowledgement

This work was supported by the Tertiary Education Commission of New Zealand's Broadband Enabled Science and Technology GRID project.

References

Antón-Canalís, L., Sánchez-Nielsen, E. and Castrillón-Santana, M., 'Fast and Accurate Hand Pose Detection for Human-Robot Interaction', *Pattern Recognition and Image Analysis*, Springer-Verlag, Berlin, LNCS 3522, pp. 553–560, 2005

Barczak, A.L.C., Dadgostar, F. and Messom, C.H., 'Real-Time Hand tracking based on non-invarient features'*, IEEE Instrumentation and Measurement Technology Conference*, pp. 2192- 2197, 2005.

Barczak, A.L., Johnson, M.J. and Messom, C.H., 'Revisiting Moment Invariants: Rapid Feature Extraction and Classification for Handwritten Digits'*, Proceedings of IVCNZ*, 2007.

Barreto, J., Menezes, P. and Dias, J., 'Human-robot interaction based on haar-like features and eigenfaces', *International Conference on Robotics and Automation*, 2004.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K. Houston, M., Hanrahan, P , 'Brook for GPUs: Stream computing on graphics hardware', *ACM Trans. Graph*. 23(3):pp. 777–786, 2004.

Crow, F.C., 'Summed-area tables for texture mapping', *Computer Graphics*, vol. 18, pp. 207–212, 1984.

Elsen, E., Houston, M. Vishal, V. Darve, E. Hanrahan, P. and Pande, V., 'N-Body simulation on GPUs', *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006.

Fan, Z., Qiu, F., Kaufman, A. and Yoakum-Stover, S. , 'Gpu cluster for high performance computing', *ACM / IEEE Supercomputing Conference*, 2004.

Gordon, M.I., Thies, W. and Amarasinghe, S., 'Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs', *12th Int Conf on Architectural Support for Programming Languages and Operating Systems*, pp.151-162, 2006.

Govindaraju, N.K. and Manocha, D., 'Cache-efficient numerical algorithms using graphics hardware', *Parallel Computing* 33 (2007) 663–684.

Huang, S.H. and Lai, S.H., 'Detecting Faces from Color Video by Using Paired Wavelet Features', *Conference on Computer Vision and Pattern Recognition Workshop*, Volume 5, 2004.

Kuo, K., Rabbah, R.M., and Amarasinghe, S., 'A Productive Programming Environment for Stream Computing'*, Proceedings of the Second Workshop on Productivity and Performance in High-End Computing*, 2005.

Lai, H.J, Yuen, P.C. and Feng, G.C., 'Face recognition using holistic Fourrier invariant features', *Pattern Recognition*, v. 34, pp. 95-109, 2001.

Liao, S.W., Du, Z.H., Wu, G.S., Lueh, G.Y., 'Data and Computation Transformations for Brook Streaming Applications on Multiprocessors', *Proceedings of the International Symposium on Code Generation and Optimization*, p.196-207, 2006.

Lienhart, R. and Maydt, J., 'An Extended Set of Haar-like Features for Rapid Object Detection', *IEEE ICIP 2002*, Vol. 1, pp. 900-903, 2002.

Lienhart, R., Kuranov, A. and Pisarevsky, V., 'Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection', *25th Pattern Recognition Symposium*, pp. 297-304, 2003.

Lienhart, R., Liang, L. and Kuranov, A., 'A Detector Tree of Boosted Classifiers for Real-time Object Detection and Tracking', *IEEE ICME2003*, Vol. 2, pp. 277-280, 2003.

Lozano, O.M. and Otsuka, K., 'Real-time Visual Tracker by Stream Processing Simultaneous and Fast 3D Tracking of Multiple Faces in Video Sequences by Using a Particle Filter', *J Sign Process Syst*, 2008, DOI 10.1007/s11265-008-0250-2.

Jones, M. and Viola, P., 'Fast Multi-view Face Detection', *Mitsubishi Electric Research Laborotories*, TR2003-96 July 2003.

Kölsch, M., Turk., M., 'Analysis of Rotational Robustness of Hand Detection with a Viola-Jones Detector', *IAPR International Conference on Pattern Recognition*, 2004.

Marziale, L., Richard III, G.G., Roussev, V., "Massive threading: Using GPUs to increase the performance of digital forensics tools", *Digital Investigation*, 4S, 2007, S73 – S81.

Micilotta, A. and Bowden, R., 'View-based Location and Tracking of Body Parts for Visual Interaction', *BMVC 2004*, Kingston, 2004.

Messom, C.H., Barczak, A.L., 'Classifier and Feature Based Stereo for Mobile Robot Systems', *IEEE International Instrumentation and Measurement Technology Conference*, 2008, pp. 997-1002.

Messom, C.H., Sen Gupta, G. and Demidenko, S., 'Hough Transform Run Length Encoding for Real-time Image Processing', *IEEE Transactions on Instrumentation and Measurement*, vol 56, no 3, pp 962-967, 2007.

Rowley, H., Baluja, S. and Kanade, T., 'Rotation Invariant Neural Network-Based Face Detection', *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 1998.

Schatz, M.C., Trapnell, C., Delcher, A.L. and Varshney, A., 'High-throughput sequence alignment using Graphics Processing Units', *BMC Bioinformatics*, 8:474, 2007.

Scheuermann, T., Hensley, J., 'Efficient Histogram Generation Using Scattering on GPUs', *ACM I3D 2007 conference proceedings*.

Stenger, B., Thayananthan, A., Torr, P., and Cipolla, R., 'Hand Pose Estimation using Hierarchical Detection', *ECCV Workshop on HCI 2004*, LNCS, Springer-Verlag, vol. 3058, pp. 102-112.

Tarditi, D., Puri, S. and Oglesby, J., 'Accelerator: Using data-parallelism to program GPUs for general-purpose uses', *12th Int Conf on Architectural Support for Programming Languages and Operating Systems*, pp. 325–335, 2006.

Viola P. and Jones, M., 'Rapid Object Detection Using a Boosted Cascade of Simple Features', *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Vol. 1, pp. 511-518, 2001.

Viola P. and Jones, M., 'Robust real-time object detection', *Second International Workshop on Theories of Visual Modelling, Learning, Computing, and Sampling*, 2001.

Wachs, J., Stern, H., Edan, Y., Gillam, M., Feied, C., Smith, M., and Handler, J., 'A Real-Time Hand Gesture System based on Evolutionary Search', *Genetic and Evolutionary Computation Conference*, 2005.

Yamagiwa, S. and Sousa, L., 'Design and implementation of a stream-based distributed computing platform using graphics processing units', *ACM Int Conf on Computing Frontiers*, pp. 197–204, 2007.