Published in Image Processing On Line on 2014–12–16.
Submitted on 2013–01–08, accepted on 2014–06–10.
ISSN 2105–1232 © 2014 IPOL & the authors CC–BY–NC–SA
This article is available online with supplementary materials,
software, datasets and online demo at
http://dx.doi.org/10.5201/ipol.2014.57

Integral Images for Block Matching

Gabriele Facciolo, Nicolas Limare, Enric Meinhardt

CMLA, ENS Cachan, France ({facciolo,nicolas.limare,enric.meinhardt}@cmla.ens-cachan.fr)

Abstract

The *integral image* representation is a remarkable idea that permits to evaluate the sum of image values over rectangular regions of the image with four operations, regardless of the size of the region. It was first proposed under the name of *summed area table* in the computer graphics community by Crow'84, in order to efficiently filter texture maps. It was later popularized in the computer vision community by Viola & Jones'04 with its use in their real-time object detection framework. In this article we describe the integral image algorithm and study its application in the context of block matching. We investigate tradeoffs and the limits of the performance gain with respect to exhaustive block matching.

Source Code

The source code and the online demo are accessible from the IPOL web page of this article[1].

Keywords: stereo; block matching; integral image; summed area table; generalized integral image

1 Introduction

The integral image representation was introduced [10, 5] with the purpose of evaluating sums of image values over axis aligned rectangular regions in constant time. Crow [5] applied it in graphics (under the name of *summed area table*) for efficiently computing *mipmaps* [26] by filtering the images with spatially varying rectangular filters. Lewis [17] used integral images for accelerating template matching. Viola and Jones [24] popularized it in the computer vision community with its application in their *real-time object detection framework*. More recently, Bay et al. [1] took advantage of integral images for the purpose of rapidly computing approximations of SIFT features with their Speeded Up Robust Features (SURF).

Block-matching aims at identifying corresponding blocks (or patches) between two images. A patch is a rectangular portion of an image. For each pixel of the first image (which we shall call from now on reference image) we need to compute the *matching cost* (denoted also *patch distance*) between the patch centered at this reference pixel, and all the patches of the second image within a given range around the reference pixel. For each patch in the reference image, the patch in the

[1] http://dx.doi.org/10.5201/ipol.2014.57

second image yielding the minimum matching cost is called *"the corresponding block"*. A popular matching cost is computed as the sum of squared differences (SSD) of pixel values within the patch.

The correspondences and matching costs provide valuable information about the similarity between the patches, which can be applied to image denoising [3], video compression [15], motion and depth estimation [21] among others. One of the first uses of integral image applied to block matching appears in the work by Faugeras et al. [8]. In that work the authors describe an optimization to spare redundant computations, that is much in the spirit of integral images. Later, Veksler [23] applied the integral images to stereo block matching for efficiently computing matching costs with multiple window sizes, and Wang et al. [25] used integral images for accelerating the patch comparison in the NL-means denoising algorithm [3].

The naive evaluation of the matching cost for a set of M relative offsets, using patches of size r^2, entails $O(Mr^2)$ operations for each patch in the reference image. Therefore, for an image with N pixels, the naive computation of all the matching costs requires $O(MNr^2)$ operations. Note that the patch size affects directly the computational cost. Many techniques have been proposed to reduce this computational load by avoiding unnecessary comparisons, for instance exploiting the regularity of the offset fields [20]. Covering all these techniques is well beyond the scope of this work. Here we are going to describe how the integral image can be applied to compute the matching costs with $O(MN)$ operations. Thus, with a cost only proportional to the number of offsets, regardless of the window size. This performance gain comes at the expense of some flexibility, a tradeoff that will be discussed in detail.

We start by describing the integral image computation and its evaluation in Section 2. In Section 3 we explain how to apply the integral image for block matching and comment some issues. Sections 4 and 5 conclude by highlighting the benefits and limitations of the integral image applied to block matching.

2 Algorithm for Computing Integral Images

For a discrete image i its integral image ii at the pixel (x, y) is defined as the sum of the pixel values of i above and to the left of (x, y), namely

$$ii(x, y) = \sum_{p \le x, q \le y} i(p, q). \tag{1}$$

Given the integral image, the sum of all the pixels within a rectangle aligned with the image axes can be evaluated with four array references, regardless of the size of the rectangle. This is illustrated in Figure 1, where the sum over the rectangle $[x_0, x] \times [y_0, y]$ is computed as $L_4 + L_1 - (L_2 + L_3)$, with $L_1 = ii(x_0 - 1, y_0 - 1)$, $L_4 = ii(x, y)$, $L_3 = ii(x_0 - 1, y)$ and $L_2 = ii(x, y_0 - 1)$.

In [24] the authors describe an algorithm for computing the integral image in one-pass over the image by using the recurrence relation

$$\begin{aligned} s(x, y) &= s(x, y - 1) + i(x, y) \\ ii(x, y) &= ii(x - 1, y) + s(x, y), \end{aligned}$$

where $s(x, y)$ is the column-wise cumulated sum, and $s(x, -1) = ii(-1, y) = 0$. Algorithm 1 details the implementation of this recurrence, where the image s is replaced with a scalar accumulator storing the current row-wise sum. A performance comparison of this method with two other slower algorithm variants is provided in Appendix A. Algorithm 2 details how to compute the sum of a rectangular region of the image by evaluating the integral image.

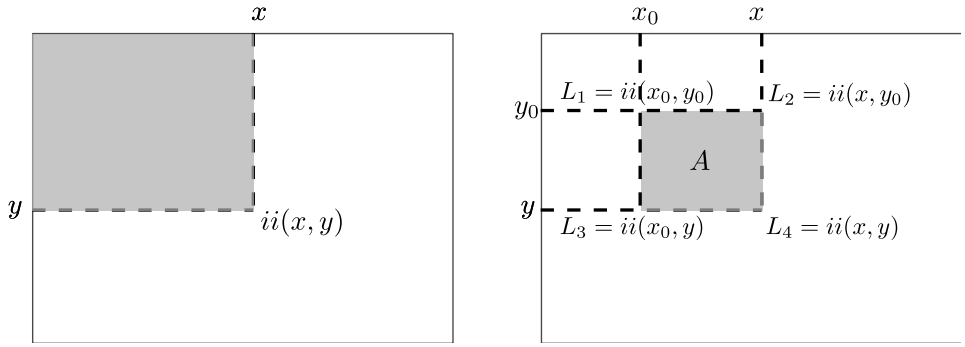


Figure 1: **Integral image and evaluation over a rectangle.** The diagram illustrates the computation of the integral image: the value of $ii(x, y)$ is the sum of the pixels in the gray region, where the upper left corner of the image is the origin of the image's coordinate system. The diagram on the right shows how to compute the sum of the samples of the discrete image on a rectangle with integer bounds, $A = [x_0, x] \times [y_0, y]$ using the integral image ii. It is enough to obtain the values of the integral image at $L_1 = ii(x_0 - 1, y_0 - 1)$, $L_4 = ii(x, y)$, $L_3 = ii(x_0 - 1, y)$ and $L_2 = ii(x, y_0 - 1)$, and to compute $L_4 + L_1 - (L_2 + L_3)$.

Algorithm 1: Computing *Integral Image*.

input: An image of size $nx \times ny$: i
output: The integral image of size $nx \times ny$: ii

$ii(0, 0) \leftarrow i(0, 0)$;
for $x \leftarrow 1$ **to** $nx - 1$ **do**
  $ii(x, 0) \leftarrow ii(x - 1, 0) + i(x, 0)$;
for $y \leftarrow 1$ **to** $ny - 1$ **do**
  $s \leftarrow i(0, y)$; `/* scalar accumulator */`
  $ii(0, y) \leftarrow ii(0, y - 1) + s$;
  for $x \leftarrow 1$ **to** $nx - 1$ **do**
    $s \leftarrow s + i(x, y)$;
    $ii(x, y) \leftarrow ii(x, y - 1) + s$;

Algorithm 2: Summing the pixel values in a rectangle of an image using the *Integral Image*.

input: The integral image for an image i: ii. Position, width (w) and height (h) of a rectangle contained in the image: $[x_0, x_0 + w - 1] \times [y_0, y_0 + h - 1]$.
output: Sum of the pixel values of i in the rectangle of size $w \times h$: S

$S \leftarrow ii(x_0 + w - 1, y_0 + h - 1)$;
if $x_0 > 0$, **then** $S \leftarrow S - ii(x_0 - 1, y_0 - 1 + h)$;
if $y_0 > 0$, **then** $S \leftarrow S - ii(x_0 - 1 + w, y_0 - 1)$;
if $x_0 > 0$ **and** $y_0 > 0$, **then** $S \leftarrow S + ii(x_0 - 1, y_0 - 1)$;

2.1 Interpretation

The simplest interpretation of the integral image comes from considering a 1D discrete signal l of length N. The sum over any interval of the signal l (for instance from x_0 to x_1) rewrites as

$$\sum_{j=x_0}^{x_1} l(j) = \sum_{j=0}^{x_1} l(j) - \sum_{j=0}^{x_0-1} l(j) = ll(x_1) - ll(x_0 - 1), \tag{2}$$

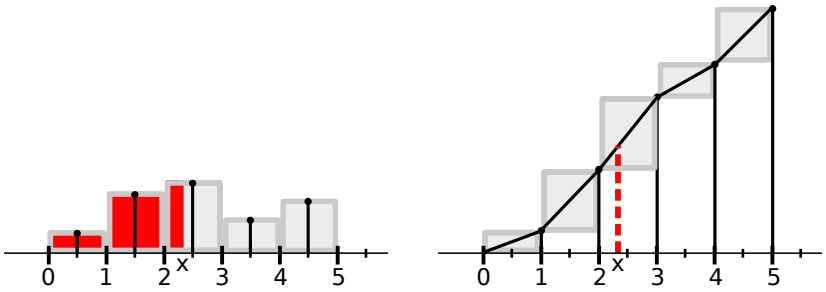


Figure 2: **Interpolating an integral image.** The left image depicts a row of pixels and the right one its associated integral line. Note, on the left image, that the samples are interpolated with nearest neighbor and that the signal is displaced half-pixel to the right. With this convention the integral image at an integer position coincides with the integral of the piecewise constant interpolation of the samples (for instance the integral image at 2 accumulates the areas of the first and second pixels). This also implies that linearly interpolating the integral image yields the integral of the piecewise constant interpolation of the samples.

where $ll(x) = \sum_{i=0}^{x} l(x)$ is the "integral line". Thus given ll, any sum over an interval of l can be computed with two lookups of ll.

It is possible to compute sums over non integer intervals by interpolating the integral line [5]. Indeed, Figure 2 illustrates a 1D discrete signal which we interpret as a piecewise constant function and the corresponding integral line. Linearly interpolating the integral line permits to integrate the underlying piecewise constant function over any (non integer) interval.

In two dimensions the integral image ii at any position (x, y) is the sum of all the pixels in the rectangular region above and to the left of (x, y) as illustrated in Figure 1. The orientation of the summed rectangle determines the "privileged orientation" for evaluating the integral image. This implies that the sums over any rectangle with sides parallel to the image axes can be evaluated with four lookups of ii. But for rectangles non parallel to the axes the evaluation of the integral image has a cost proportional to its perimeter. For 45° oriented rectangles, Lienhart and Maydt [18] proposed a rotated integral image. The rotated integral image is computed traversing the image in the diagonal directions. The sums over rotated rectangles are then computed, like the integral image, with four table lookups.

Regarding the sum of values of a signal f as a convolution with a box filter g entails that any invertible linear operator can be applied to f if its inverse is applied to g (for commodity we switch to continuous notation). Concretely with the derivative/integral pair we have (in 1D)

$$(f * g)(x) = \int f(t)g(x-t)dt = \int F'(t)g(x-t)dt = \int F(t)g'(x-t)dt, \tag{3}$$

where $F(t) = \int_{-\infty}^{t} f(t')dt'$. Observing that the derivative of the box filter is the sum of two delta functions, the computation of the sum is reduced to the evaluation F at the two points. This insight was used by the authors of [22] to efficiently filter signals with a piecewise constant filter g, by converting g into a sequence of deltas at its discontinuities. The same computation extends to two (and more) dimensions by simply taking partial derivatives and integrals on each dimension:

$$(f * g)(x, y) = \iint F(t, s)(\frac{d}{dy}\frac{d}{dx}g(x-t, y-s))dtds, \qquad \text{with} \qquad F(t, s) = \int_{-\infty}^{s}\int_{-\infty}^{t} f(t', s')dt'ds'.$$

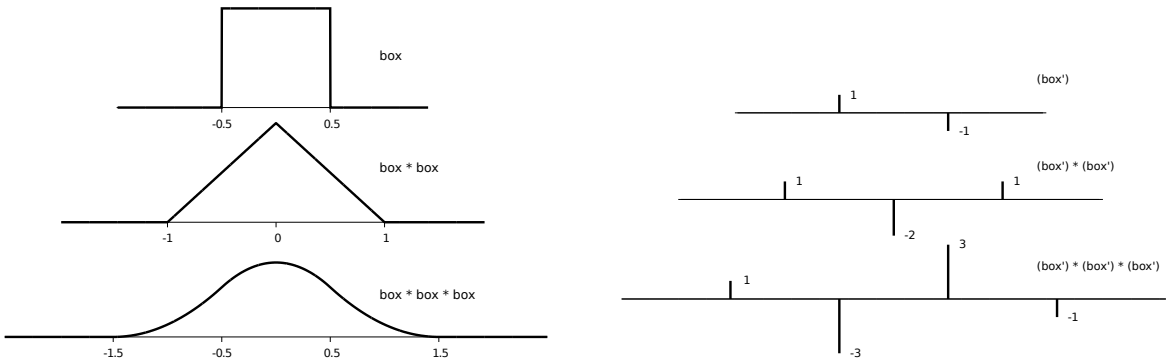


Figure 3: **B-splines of order n.** The order n B-spline is obtained by convolving the box filter $n-1$ times with itself, here shown for $n = \{1, 2, 3\}$. The n-th derivative of an order n B-spline is a sequence of deltas depicted on the right. Note that the B-spline of order 3 is a good approximation of the Gaussian.

n=0	n=1	n=2	n=3
1	-1 1 1 -1	1 -2 1 -2 4 -2 1 -2 1	-1 3 -3 1 3 -9 9 -3 -3 9 -9 3 1 -3 3 -1

Figure 4: **Coefficients for 2D B-spline filtering using generalized integral images.** The 2D coefficients for filtering with an n-th order B-splines evaluating the corresponding n-fold integral image. Adjusting the spacing between the samples yields filters of different widths. The zero order corresponds to sampling the image with a delta.

2.2 Generalized Integral Images

An alternative proof of the property (3) expresses the integration and derivation using the Fourier transform $f(x) \longleftrightarrow \hat{f}(\omega)$ [12]. The spatial integration and derivation of a function correspond to division and multiplication by $i\omega$ in the frequency domain, that is: $\frac{df}{dx}(x) \longleftrightarrow i\omega\hat{f}(\omega)$ and $\int f(x) \longleftrightarrow \hat{f}(\omega)/(i\omega)$. Hence we have the chain of equivalent computations

$$f(x) * g(x) \longleftrightarrow \hat{f}(\omega)\hat{g}(\omega) = \frac{\hat{f}(\omega)}{(i\omega)^n}\,(i\omega)^n\hat{g}(\omega) \longleftrightarrow \left(\int^n f\right)(x) * \left(\frac{d^n g}{dx^n}\right)(x),$$

where $\int^n f$ and $\frac{d^n g}{dx^n}$ denote respectively the n-fold integration and n-th order derivative of the signals.

This observation leads to a generalization of the integral images [12, 9, 7] to n-fold integral images, which allows to filter signals by piecewise polynomial filters in constant time (regardless of the filter's support). By increasing the integral order it is possible to define smoother and more isotropic filters, which in turn reduce the artifacts common with the box filter.

As shown in Figure 3, convolving n box filters generates the n-th order B-spline [12, 7], whose n-th derivative is a sequence of deltas. Therefore, given the n-fold integral of an image it is possible to evaluate its convolution with a B-spline of the same order, by evaluating a few samples. In Figure 4 the coefficients of several low order 2D B-splines are shown. Adjusting the spacing between the samples yields the filters of different widths.

There are a couple of issues to keep in mind when working with n-fold integral images. Firstly, the representation precision (in bits per pixel) of an n-fold integral image increases linearly with n. This means that storing the n-fold integral image of a signal of length 2^m and with b bits per sample

demands an integral image with a precision of $n \times m + b$ bits. For instance, the pixels of the 1-fold integral image are represented with $m + b$ bits. Remarking that a 2-fold integral image accumulates 2^m values from the 1-fold image, it follows that its maximum range is $2^{m+b}\,2^m$, which amounts to $2 \times m + b$ bits. Secondly, in 2D the cost of evaluating an n-fold integral image grows quadratically with n. Thus for instance, applying an order 3 B-spline requires 16 accesses to the integral image (see Figure 4), while the order 1 B-spline (the rectangle function) only requires 4. This means that for small blocks (smaller than 7×7, according to Table 1) it is more effective to compute the sums exhaustively without recurring to integral images. For more details about the generalized integral images we recommend [5, 12, 7] as further reading.

2.3 Numerical Precision Notes

The numerical precision of the integral image depends on the concrete data being accumulated, and the precision of the data type used to store the image. In this section we analyze the precision limits of the integral image representation for some typical cases. Experiments with different representation precisions are performed by adapting[2] the integral image code associated to this article[3].

- **Integer pixel values and arithmetic overflow.** The simplest situation is the accumulation of b-bit integers into an m-bit data type. In this case one can accurately accumulate at most $2^{(m-b)}$ values. As an example, if the data vector contains 12-bit unsigned integers and the integral image is stored on a 32-bit unsigned integer data type, then only $2^{(32-12)} = 1024 \times 1024$ values can be accumulated without any risk of an integral value becoming larger than the 32-bit integer capacity and resulting in an integer overflow[4]. Stored on 64-bit data, the integral image can safely hold the much reassuring quantity of $2^{52} = 4 \times 10^{15}$ values.

- **Fixed point pixel values and precision loss.** Usually we compute the square (SSD for instance) of pixels with fixed point values. In that case the range of the data is duplicated. For instance, the square of a 16-bit fixed point value has a range of 32-bits. A single precision floating-point data type (IEEE754 32-bits) has only 23 bits of mantissa and is not sufficient here. A double precision integral image (IEEE754 64-bit floating-point, 53-bit mantissa) can hold 2 million (or 2^{53-32}) of such 32-bit values without precision loss. As a reference a 1400×1400 pixels image contains 2 million pixels. With more values to be stored, the least-significant bits of the mantissa in the integral image values are at risk of being dropped when the integral values grow. These errors are relatively small when compared with the sum but will become significant when the difference between two close locations of the integral image is computed ($ll(x_1) - ll(x_0 - 1)$ in equation (2)).

  Figure 5 , second row, illustrates the precision of the integral image evaluation for a large (2048×2048 pixels) synthetic image, for several different data types. The pixel values of this image are in the range $[2^{-19}, 2^{19}]$. This type of range could be the result of another processing with fixed point precision. Note how in Figure 5, for the single precision case, the error grows with the distance from $(0, 0)$. This is due to the accumulation of truncation errors.

[2] In the code associated to this article, the preprocessor macro `IIFLOAT` and custom type name `iifloat_t` define the data type used for representing the integral image. By default it is set to extended precision with `-DIIFLOAT='long double'` passed in the compiler options.

[3] http://dx.doi.org/10.5201/ipol.2014.57

[4] In most situations an integer overflow results in the value being wrapped and stored modulo 2^{32}, which still allows for some arithmetic operations to be performed on this data when we are aware of the possibility of an overflow. But in general the consequences of an overflow are not safely predictable from the programming language standard alone [16, section A7] and this situation should be avoided.

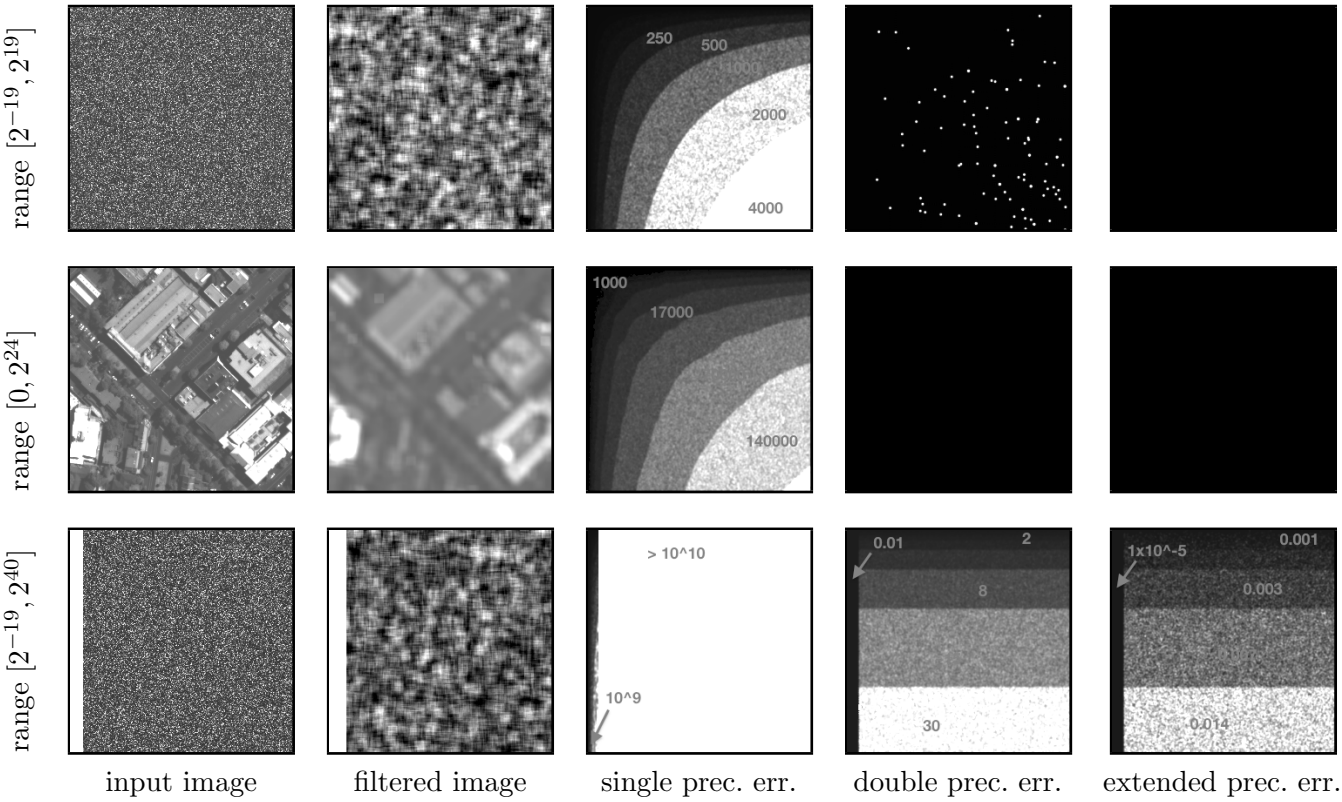


Figure 5: **Precision of the integral image for single, double and extended types.** From left to right, the first column corresponds to an extract from a 2048×2048 image to be filtered, the second column shows the image obtained by computing the sum over 11×11 patches of the first image. The third column shows the error (for the full 2048×2048 image) resulting from the truncation of a single precision integral image, the labels indicate the averaged error values. The forth and fifth columns show the errors produced by double precision and extended precision integral images.

The image in the first row corresponds to synthetic data in the range $[2^{-19}, 2^{19}]$. The concrete image corresponds to squaring i.i.d. Gaussian noise. For this image the single precision representation is clearly insufficient. With double precision the maximum error is 0.25 (corresponding to the white dots), and for extended precision the error is 0 everywhere.

In the second row we repeated the experiment with a natural image with values in the range $[0, 2^{24}]$ (squared 12-bit integers). Again the single precision representation is insufficient, while double precision and extended precision produce no errors.

The experiment in the third row corresponds to a "nightmare" scenario for the precision of the integral image representation. The image is similar to the one of the first experiment (range $[2^{-19}, 2^{19}]$), except that 2^{40} is added to the values of the first 100 columns (saturated in the image) making the range of the whole image $[2^{-19}, 2^{40}]$. For the single precision representation the errors are always above 10^7. Observe that although the largest values are in the first columns of the image the precision there is always relatively small compared to the rest of the image. This is due to the truncation induced by the large values accumulated in the first band.

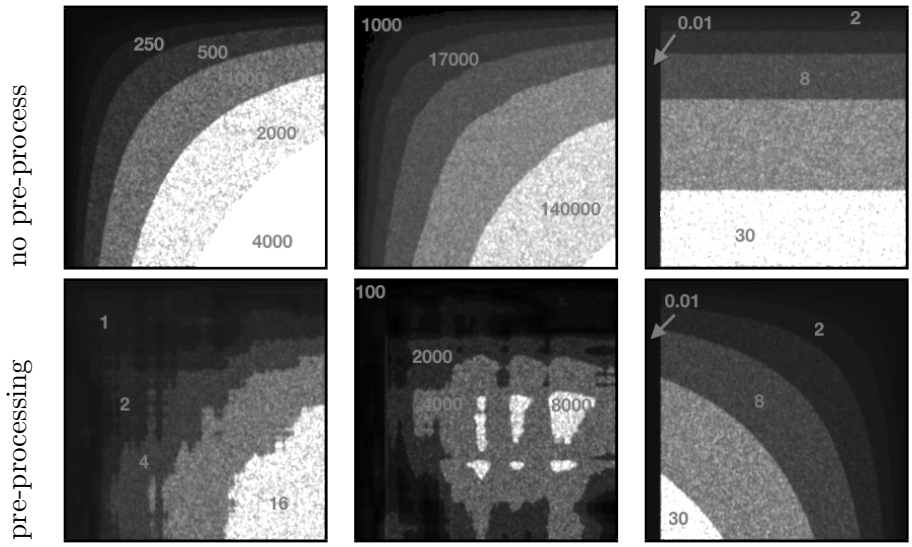


Figure 6: **Removing the average increases the precision.** Each column corresponds to one experiment of Figure 5: synthetic image (single precision), natural image (single precision) and nightmare scenario (double precision). The first row shows the error from Figure 5, while the second row shows the error when the input data have been pre-processed by removing the average value of the image. In general this pre-processing reduces the truncation errors. In the third column note that although the error is more concentrated, its maximum value is the same.

- **Nightmare scenario.** In practice the floating point data types are used to represent all non integer values, because they can be used to represent very large and very small numbers. But what matters for the integral image is the fixed point precision of its data type, namely the number of evenly spaced values the data type can represent in a certain range. The nightmare scenario for the integral image is to accumulate floating point data that spans an extreme range of floating point values, for example 2^{-100} to 2^{100}. Indeed, storing this data accurately at fixed point precision requires more than 200 bits per pixel. The third row of Figure 5 shows an example of this scenario, with an image containing floating point values in the range $[2^{-19}, 2^{19}]$, but where we added 2^{40} to the pixels of the first 100 columns (saturated on the images), thus the range of the whole image is $2^{40+19} = 2^{59}$. For this case the double precision is insufficient to store the integral image without error. Also note that, although the first columns contain the largest image values, the precision there remains acceptable, while it deteriorates over the portion of the image with smaller values. This is the result of the truncation induced by the large values accumulated in the first band.

Although floating point data may easily surpass the integral image's precision, usual data do not span such large ranges of precisions, so one could still apply the integral image in most cases. As a rule of thumb, we can say that single precision integral images should NEVER be used, unless being very sure of not incurring in precision errors.

Preprocessing by removing the average value. Removing the average value from the data increases the precision of the representation as seen in Figure 6. However, this improvement may be less important for some images. The cost of this pre-processing is one extra sweep through the data to compute the average, and one extra addition for each evaluation of the integral image.

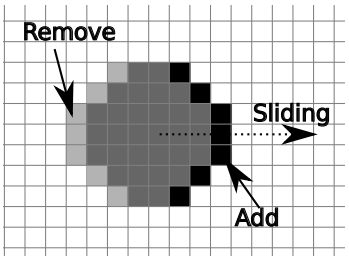


Figure 7: **Rolling sums in 2D.** Updating the boundary values while sliding the window permits to filter the data with any window shape.

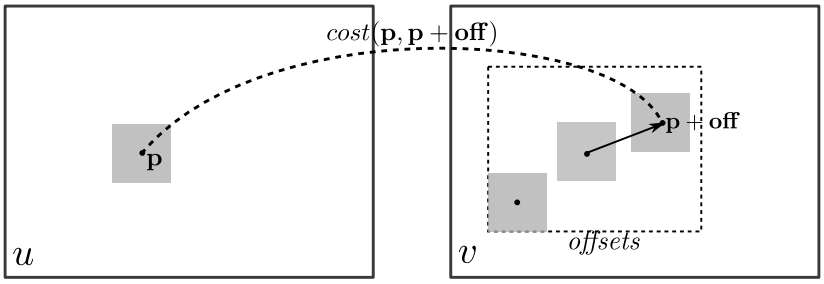


Figure 8: **Block matching.** The objective of block matching is to compute the matching costs between every patch of the reference image (centered at $\mathbf{p}$) and all the patches of the second image centered at $\mathbf{p} + \mathbf{off}$, where $\mathbf{off} \in \mathit{offsets}$.

The case of *rolling sums*. A frequent special case worth a special mention is when the sums are computed on windows of fixed size $r \times r$. In this case instead of computing the integral image, it is possible to filter the data by using rolling sums with much less risk of precision loss. For a 1D signal $i(\cdot)$ the filtered signal $Si(\cdot)$ is computed with rolling sums as

$$Si(x) \leftarrow Si(x-1) + i(x) - i(x-r-1) \qquad \forall x \in [0 \dots nx-1],$$

where $i(x) = 0$ for $x < 0$. Figure 7 illustrates the rolling sums in 2D. Note that it doesn't really compute the integral image, but in the same spirit [8] it recycles the computations performed for the previous window.

3 Block Matching using Integral Images

In this section we describe an algorithm for efficiently performing block matching using integral images. We then specify the matching costs that can be implemented using integral images. Lastly we analyze the performance of this algorithm with respect to the exhaustive search and Fourier correlation for different window sizes.

The objective is to compute the matching costs between patches of two images (which may be the same image). For each pixel of the reference image, we want to compute the matching cost between the patch centered on this reference pixel and all the patches of the second image centered at pixels within a given range around the reference pixel, as illustrated in Figure 8. Let us denote $u, v : \Omega \to \mathbb{R}$ the reference and secondary images respectively, where $\Omega \subset \mathbb{Z}^2$ denotes the discrete image domain, and positions in $\mathbb{Z}^2$ are boldfaced, $\mathbf{p} = (x, y)$.

We consider matching costs that can be expressed as sums of distances between values at pixels $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}^+$. For instance, taking the pixel value distance $d(x, y) = |x - y|^2$ summed over patches

of size r^2 yields the matching cost known as SSD (Sum of Squared Differences)

$$\mathrm{SSD}(\mathbf{p},\mathbf{q}) = \sum_{\mathbf{t}\in B(0,r)} d(u(\mathbf{p}+\mathbf{t}), v(\mathbf{q}+\mathbf{t})) = \sum_{\mathbf{t}\in B(0,r)} |u(\mathbf{p}+\mathbf{t}) - v(\mathbf{q}+\mathbf{t})|^2,$$

where $B(0,r)$ denotes the patch of side r centered at 0. Supposing that the size of the images is $|\Omega| = N$ pixels, a naive evaluation of the matching costs for a set of M offsets entails (by the algorithm shown hereafter) $O(NMr^2)$ operations.

```
1 for p ∈ u do
2     for off ∈ offsets do
3         cost ← 0;
4         for t ∈ B(0, r) do
5             cost ← cost + d(u(p + t), v(p + off + t));
```

Note, however, that the evaluated patches are overlapping, therefore many of the computations are redundant. Particularly, if the pixels $\mathbf{p}$ are contiguous then each pixel is included in r^2 patches, so the same pixel value distance intervenes in the computation of r^2 matching costs.

The key of the integral image approach to block matching is to rewrite the previous algorithm so that each pixel value distance is computed only once, and reused in all of the r^2 patches, yielding a cost of $O(MN)$. This is shown in the algorithm hereafter.

```
1 for off ∈ offsets do
2     for p ∈ u do
3         dist(p) ← d(u(p), v(p + off))      /* precompute pixel distances          */
4     ii ← ComputeIntegralImage(dist)        /* cost ∼ 2 additions per pixel        */
5     for p ∈ u do
6         cost ← SumPixelsEvaluatingIntegralImage(ii, p, r);
```

Concretely, note how the offset loop is placed outside the pixel loop. This implies that all the costs associated to a single offset are evaluated simultaneously. Then, for each offset all the pixel value differences are computed, and an integral image is produced from them, which in turn permits to compute the sums in constant time.

3.1 Matching Costs That Can Be Implemented with an Integral Image

Integral images can be used to compute any matching cost which can be expressed as a sum of pixel value distances d, provided that these distances are independent of the pixel positions $\mathbf{p}$ and $\mathbf{q}$:

$$cost(\mathbf{p},\mathbf{q}) := \sum_{\mathbf{t}\in B(0,r)} d(u(\mathbf{p}+\mathbf{t}), v(\mathbf{q}+\mathbf{t})).$$

Here we list some well-known costs that can be reduced to sums of independent pixel-wise distances, and therefore be efficiently implemented using integral images.

1. **SSD and SAD.** These costs are obtained by taking $d(a,b) = |a-b|^p$. Then for $p = \{1,2\}$ we have the *Sum of Absolute Differences* (SAD) and the *Sum of Squared Differences* (SSD) respectively. But any value of p will work as well.

2. **ZSSD.** The *Zero-mean SSD* adds invariance with respect to additive contrast changes to the SSD. It is defined by

$$\mathrm{ZSSD}(\mathbf{p},\mathbf{q}) := \sum_{\mathbf{t}\in B(0,r)} \left|(u(\mathbf{p}+\mathbf{t}) - \mu_u(\mathbf{p})) - (v(\mathbf{q}+\mathbf{t}) - \mu_v(\mathbf{q}))\right|^2,$$

where $\mu_v(\mathbf{q}) = \frac{1}{|B(0,r)|}\sum_{\mathbf{t}\in B(0,r)} v(\mathbf{q}+\mathbf{t})$ is the precomputed average of pixel values in v over the block centered at $\mathbf{q}$. The ZSSD cannot be implemented with integral images if expressed in this form, because the pixel value distances are dependent of the position of the pixels. Indeed, depending on the position, a different pair of means μ_v and μ_u are being subtracted. However, expanding the square we get

$$\mathrm{ZSSD}(\mathbf{p},\mathbf{q}) := \mathrm{SSD}(\mathbf{p},\mathbf{q}) - |B(0,r)|\,(\mu_v(\mathbf{q}) - \mu_u(\mathbf{p}))^2$$

and since $\mu_v(\cdot)$ and $\mu_u(\cdot)$ can be precomputed (also with integral images), the overall cost of ZSSD is comparable to SSD.

3. **ZSAD.** The *Zero-mean SAD* [13] cannot be implemented using integral images, because the corresponding pixel distance $d(u(\mathbf{p}+\mathbf{t}), v(\mathbf{q}+\mathbf{t})) = |u(\mathbf{p}+\mathbf{t}) - v(\mathbf{q}+\mathbf{t}) + \mu_v(\mathbf{q}) - \mu_u(\mathbf{p})|$ is not independent of the current patch centers ($\mu_v(\mathbf{q})$ and $\mu_u(\mathbf{p})$ depend on the patch center).

   However, **a similar distance can be computed by pre-filtering the input images**. Following the terminology introduced in the work by Hirschmüller and Scharstein [13] we call this method *mean filter*. It amounts to precompute $\tilde{u} = u - \mu_u$ and $\tilde{v} = v - \mu_v$. Then compute the SAD cost by taking $\tilde{u}$ and $\tilde{v}$ instead of u and v. The same technique could also be used to approximate ZSSD.

4. **SSDNorm.** Normalizing the patches by their L^2-norms renders the comparison robust to multiplicative contrast changes. The SSD/Norm is defined as

$$\mathrm{SSD/Norm}(\mathbf{p},\mathbf{q}) := \sum_{\mathbf{t}\in B(0,r)} \left|\frac{u(\mathbf{p}+\mathbf{t})}{\|u|_{B(\mathbf{p},r)}\|} - \frac{v(\mathbf{q}+\mathbf{t})}{\|v|_{B(\mathbf{q},r)}\|}\right|^2,$$

   where $\|u|_{B(\mathbf{p},r)}\| = \sqrt{\sum_{\mathbf{t}\in B(0,r)} (u(\mathbf{p}+\mathbf{t}))^2}$. Expanding the above expression we get a correlation formula

$$\mathrm{SSD/Norm}(\mathbf{p},\mathbf{q}) := 2 - 2\frac{\mathrm{Prod}(\mathbf{p},\mathbf{q})}{\|u|_{B(\mathbf{p},r)}\|\ \|v|_{B(\mathbf{q},r)}\|},$$

   where $\mathrm{Prod}(\mathbf{p},\mathbf{q}) := \sum_{\mathbf{t}\in B(0,r)} u(\mathbf{p}+\mathbf{t})v(\mathbf{q}+\mathbf{t})$, which can be computed using integral images by taking $d(a,b) = ab$, and the terms $\|u|_{B(\mathbf{p},r)}\|$ and $\|v|_{B(\mathbf{q},r)}\|$ can be precomputed.

   The implementation of this cost should prevent divisions by zero. Under this circumstance, the cost should be set to 2.

5. **NCC.** The *Normalized Cross Correlation* combines the benefits of ZSSD and SSDNorm as it is invariant to affine contrast changes. It is defined as

$$\mathrm{NCC}(\mathbf{p},\mathbf{q}) := 1 - \mathrm{Corr}(\mathbf{p},\mathbf{q}),$$

   with

$$\mathrm{Corr}(\mathbf{p},\mathbf{q}) := \frac{\frac{1}{|B(0,r)|}\sum_{\mathbf{t}\in B(0,r)} (u(\mathbf{p}+\mathbf{t}) - \mu_u(\mathbf{p}))\,(v(\mathbf{q}+\mathbf{t}) - \mu_v(\mathbf{q}))}{\sqrt{\sigma^2\left(u|_{B(\mathbf{p},r)}\right)\ \sigma^2\left(v|_{B(\mathbf{q},r)}\right)}}, \tag{4}$$

and where $\sigma^2\left(u|_{B(\mathbf{p},r)}\right)$ is the sample variance of the block centered at $\mathbf{p}$.

Corr() takes values in $[-1,1]$, where 1 indicates the maximum correspondence. For a consistent notation across the different methods NCC is defined as $1-\text{Corr}$. The above expression can be written as

$$\text{Corr}(\mathbf{p},\mathbf{q}) := \frac{\frac{1}{|B(0,r)|}\text{Prod}(\mathbf{p},\mathbf{q}) - \mu_u(\mathbf{p})\cdot\mu_v(\mathbf{q})}{\sqrt{\sigma^2\left(u|_{B(\mathbf{p},r)}\right)\,\sigma^2\left(v|_{B(\mathbf{q},r)}\right)}},$$

which can be computed using one integral image (for Prod()) for each offset value $(\mathbf{q}-\mathbf{p})$, and where the terms μ and σ^2 can be precomputed by integral images too.

The implementation of this cost must prevent divisions by zero. Under that circumstance the cost should be set to 1.

6. **AFF.** The *"affine" similarity measure* [6] is designed to match patches independently of affine contrast changes, but unlike the NCC it can distinguish flat patches from those containing edges. It can be seen from Equation (4) that Corr becomes arbitrarily small when one of the patches has low variance, which means that the distance between any patch and a quasi-flat patch will be small. In contrast, the *"affine" similarity measure* [6] defined below can distinguish patches under these circumstances:

$$\text{AFF}(\mathbf{p},\mathbf{q}) := \max\left(\min_{\alpha\geq 0,\beta}\|U_{\mathbf{p}}-\alpha V_{\mathbf{q}}-\beta\|\,,\min_{\alpha\geq 0,\beta}\|V_{\mathbf{q}}-\alpha U_{\mathbf{p}}-\beta\|\right),$$

where $U_{\mathbf{p}} = u|_{B(\mathbf{p},r)}$ and $V_{\mathbf{q}} = v|_{B(\mathbf{q},r)}$. It can be explicitly computed by the formula

$$\text{AFF}^2(\mathbf{p},\mathbf{q}) := \max\left(\sigma^2\left(u|_{B(\mathbf{p},r)}\right),\sigma^2\left(v|_{B(\mathbf{q},r)}\right)\right)\cdot\min\left(1,1-\text{Corr}(\mathbf{p},\mathbf{q})|\text{Corr}(\mathbf{p},\mathbf{q})|\right),$$

which can be implemented with integral images after precomputation of $\sigma^2\left(u|_{B(\mathbf{p},r)}\right)$ and $\sigma^2\left(v|_{B(\mathbf{q},r)}\right)$, which in turn can be done with two integral images, one for the square of u and one for u. Note that, unlike the NCC, the value of the *"affine" similarity measure* is not itself invariant to affine contrast changes. This implies that the cost associated to a pair of bright patches is higher than the cost associated to a pair of dark patches with the same structure as the bright pair.

7. **LIN.** This is a simpler variant of the AFF cost that drops the invariance to additive changes, but has a similar performance

$$\text{LIN}(\mathbf{p},\mathbf{q}) := \max\left(\min_{\alpha\geq 0}\|U_{\mathbf{p}}-\alpha V_{\mathbf{q}}\|\,,\min_{\alpha\geq 0}\|V_{\mathbf{q}}-\alpha U_{\mathbf{p}}\|\right).$$

It can also be implemented with integral images by re-writing it as

$$\text{LIN}^2(\mathbf{p},\mathbf{q}) := \max(\|U_{\mathbf{p}}\|^2,\|V_{\mathbf{q}}\|^2)\left(1-\frac{[\text{Prod}(\mathbf{p},\mathbf{q})]^2}{\|U_{\mathbf{p}}\|^2\|V_{\mathbf{q}}\|^2}\right).$$

The implementation of AFF and LIN costs must prevent divisions by zero. Under that circumstance, the cost should be set respectively to the maximum variance or norm of the two patches.

8. **BTSAD and BTSSD.** By BTSAD or BTSSD, we mean an adaptation to SAD or SSD of the Birchfield and Tomasi [2] sampling insensitive pixel dissimilarity. This matching cost, originally proposed for stereo matching, is designed to be insensitive to image sampling. For

smooth (non aliased) images this cost is proven to be stable with respect to subpixel translations of the patches, while still evaluating the costs at integer positions.

This cost is particularly useful in combination with global block matching methods (Dynamic Programming for instance), where the dissimilarity is accumulated along scanlines, and where the subpixel computations are too expensive. The insensitivity usually prevents the wrong classification of some pixels as occlusions.

The usefulness of this matching cost for a "local" block matching method is less clear, since the minimum SAD/SSD cost may not change. Nevertheless it is worth comparing its performance with subpixel matching.

The matching costs proposed by Birchfield & Tomasi replace the definition of the pixel value distances with d^{BT}. For the case of the SSD we get

$$BTSSD(\mathbf{p},\mathbf{q}) := \sum_{\mathbf{t}\in B(0,r)} |d^{BT}(I_L(\mathbf{p}+\mathbf{t}), I_R(\mathbf{q}+\mathbf{t}))|^2,$$

where the distance d^{BT} is defined as a symmetrization of the distance by

$$d^{BT}(I_L(\mathbf{p}), I_R(\mathbf{q})) = \min(\bar{d}(I_L(\mathbf{p}), I_R(\mathbf{q})), \bar{d}(I_R(\mathbf{q}), I_L(\mathbf{p}))),$$

and where $\bar{d}$ is the distance of I_L to the interval $[I_R^{min}, I_R^{max}]$

$$\bar{d}(I_L(\mathbf{p}), I_R(\mathbf{q})) = \max(0, I_L(\mathbf{p}) - I_R^{max}(\mathbf{q}), I_R^{min}(\mathbf{q}) - I_L(\mathbf{p})),$$

$$\bar{d}(I_R(\mathbf{q}), I_L(\mathbf{p})) = \max(0, I_R(\mathbf{q}) - I_L^{max}(\mathbf{p}), I_L^{min}(\mathbf{p}) - I_R(\mathbf{q})).$$

The four precomputed images $I_R^{max}, I_R^{min}, I_L^{max}, I_L^{min}$ contain the maximum and minimum interpolated values of the image in a half pixel neighbor. For instance, with I_R we have

$$I_R^{min}(\mathbf{q}) = \min(I_R^-(\mathbf{q}), I_R^+(\mathbf{q}), I_R(\mathbf{q})) \quad \text{and} \quad I_R^{max}(\mathbf{q}) = \max(I_R^-(\mathbf{q}), I_R^+(\mathbf{q}), I_R(\mathbf{q})),$$

where the interpolated values are computed by bilinear interpolation:

$$I_R^-(\mathbf{q}) = \frac{1}{2}(I_R(\mathbf{q}) + I_R(\mathbf{q} - (1,0)^T)), \quad I_R^+(\mathbf{q}) = \frac{1}{2}(I_R(\mathbf{q}) + I_R(\mathbf{q} + (1,0)^T)).$$

Note that the maximum (and minimum) of the interpolated pixels I^{max} (resp. I^{min}) are only computed along the horizontal axis (definitions of I^+ and I^-). This is because the cost was originally proposed for stereo matching, so the subpixel differences are supposed to occur only along the horizontal axis. A straightforward extension of this cost for two dimensional block matching (**BT2DSAD** and **BT2DSSD**) replaces the definition of I^{max} (resp. I^{min}) to consider subpixel offsets also in the vertical direction

$$I^{max}(\mathbf{q}) = \max\left\{\hat{I}(\mathbf{q}+\mathbf{V})\right\} \quad \text{with} \quad \mathbf{V} = [-1/2, 1/2]^2,$$

where $\hat{I}$ denotes the bilinear interpolation of the image I. For this interpolation the maximum (resp. minimum) will occur at one of nine possible positions, therefore

$$I^{max}(\mathbf{q}) = \max\left\{\hat{I}(\mathbf{q}+\mathbf{s}) : s \in \{-\frac{1}{2}, 0, \frac{1}{2}\} \times \{-\frac{1}{2}, 0, \frac{1}{2}\}\right\}.$$

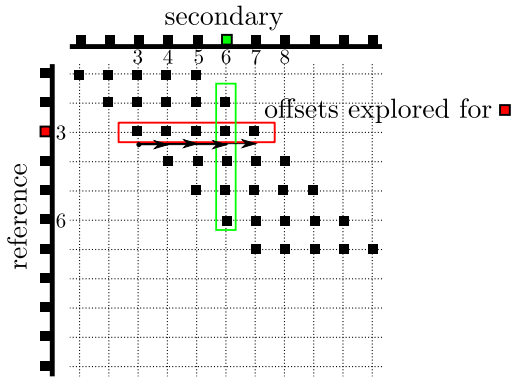


Figure 9: **1D illustration of matching and reverse matching.** The above figure shows a *disparity space diagram* representing all the possible matchings that can be computed from pixels in one line of the reference image and the corresponding line of the secondary image. The pixels of the reference image are represented in the vertical axis while the horizontal coordinates are the pixels of the secondary image. The dots on the grid represent the computed costs: for a pixel in the reference image several costs are computed corresponding to a set of 1D offsets. If the matching cost is symmetric, that is $cost(\mathbf{p}, \mathbf{q}) = cost(\mathbf{q}, \mathbf{p})$, then the computed costs also contain the matching costs from secondary to the reference image, for instance for the green pixel five negative offsets have been considered.

9. **Pre-filtering.** Hirschmüller and Scharstein in their work [13] describe a set of matching costs that amount to pre-filtering the input images. They consider: the *mean filter* which subtracts from each pixel the mean intensities within a fixed neighborhood of 15×15 pixels, the *Laplacian of Gaussian* which performs a smoothing that removes noise and intensity offsets, the *background subtraction by bilinear filtering* which smooths the image without removing high contrast textures working on windows of 15×15 pixels.

   Other types of pre-filters [13] such as the *rank filter*, *soft rank* and the *consensus filter*, cannot be implemented using integral images because they modify the pixel values on a patch basis.

It is important to remark that evaluating an integral image on a rectangle always reduces to a sum: $L_4 + L_1 - (L_2 + L_3)$, which is not guaranteed to be positive. To prevent the appearance of negative matching costs, due to the limited numerical precision, the implementation of the costs always truncates them at 0.

3.2 Block Matching Algorithm

We now detail the algorithm for performing block matching using integral images. As anticipated in the beginning of this section, the algorithm will evaluate all the matching costs for a given offset, one offset at a time (see Algorithm 3). For each offset, the costs of all the pixels $\mathbf{p}$ are computed, and at each position only the offset yielding the minimum cost is kept.

The same algorithm permits to compute the reverse matchings from the second to the reference image without overhead. If the cost is symmetric, that is $cost(\mathbf{p}, \mathbf{q}) = cost(\mathbf{q}, \mathbf{p})$, and the considered offsets are all contiguous, then the costs computed for the reference image coincide with the costs from the secondary to the reference image (considering negative offsets). This is illustrated in Figure 9. This permits to compute the reverse matchings without overhead. Conversely, if the computed offsets are not contiguous, then some of the reverse matchings may not be considered.

Roughly, the block matching algorithm as detailed in Algorithm 3 is:

- First initialize the variables containing the minimum costs, and precompute the images needed for evaluating the current cost. These images contain the patch averages, the patch variances, or patch norms, depending on the chosen cost.
- For each offset, an image of pixel-wise distances is computed: $dist(\mathbf{p}) \leftarrow d(u(\mathbf{p}), v(\mathbf{p}+\mathbf{off}))$ Summing the values of *dist* over patches yields the matching costs for the current offset. The computation of some costs may also require to access the precomputed images. The sums over patches is efficiently computed using integral images.
  - The integral image of *dist* is computed. This permits to evaluate the matching cost of each patch in constant time.
  - For each pixel of the reference image, the cost for the current offset is computed. If the cost is smaller than any previous cost, then the offset is associated to the pixel. Likewise, the costs and the (reverse) matchings corresponding to the secondary image are updated as well.

Algorithm 3: Block matching from image u to image v, and the reverse matching (v to u), using *integral images* for computing the matching costs.

input: The reference and secondary images: u and v. A patch size: r.
A set of offsets to evaluate: *offsets*.

output: The offsets of the matches from u to v and the corresponding minimum matching costs between the patches: matchOffset, minCost.
Offsets and costs for the *reverse* matches from v to u: R_matchOffset , R_minCost.

```
   /* initialize the costs                                                    */
1  minCost ← ∞ ;
2  R_minCost ← ∞ ;
   /* Precomputed images needed for the cost evaluation (μ_u, μ_v, σ_u,...)   */
3  Stuff ← Precompute_Stuff(u, v) ;
4  for off ∈ offsets do
       /* precompute pixel distances                                          */
5      for p ∈ u do
6          dist(p) ← d(u(p), v(p + off))
       /* compute integral image                                              */
7      ii ← ComputeIntegralImage(dist);
8      for p ∈ u do
           /* compute matching cost for pixel p and offset off by evaluating the
              integral image and the precomputed images                       */
9          newCost ← EvalCostWithIntegralImage( ii, p, off, Stuff );
           /* update the minimum costs                                        */
10         if minCost(p) > newCost then
11             minCost(p) ← newCost;
12             matchOffset(p) ← off;
           /* update the Reverse matching                                     */
13         if R_minCost(p + off) > R_newCost then
14             R_minCost(p + off) ← R_newCost;
15             R_matchOffset(p + off) ← −off;
```

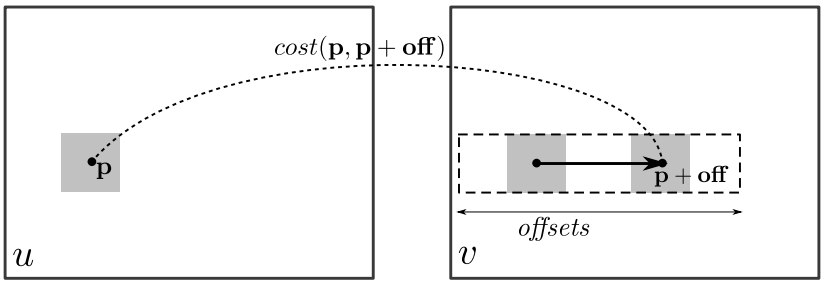


Figure 10: **Stereo Block matching.** For a stereo rectified image pair the possible correspondences are restricted to horizontal lines, therefore the stereo block matching only considers horizontal *offsets*.

A remark about parallelization. The previous algorithm can be parallelized by splitting the reference image into smaller sub-images and applying the algorithm in parallel to each sub-image. Besides reducing the computing time, this strategy also provides a way to deal with the numerical precision issues connected to the use of large integral images (as mentioned in Section 2.3). The block matching implementation associated to this work[5] processes the reference image by horizontal bands, with each band treated in parallel.

3.3 Subpixel Stereo Block Matching

The apparent motion of a point in an image due to camera displacement is related to its distance from the camera. This is the principle of stereoscopy, which allows to estimate the depths of a scene by determining the relative motion (parallax movement) of corresponding points in two images. In the pinhole camera geometry it can be shown that the parallax movement of a point occurs only along a line (called epipolar line), and that the image pairs can always be *rectified* (transformed by an homography) so that the apparent motion of pixels is horizontal everywhere [11]. The apparent motion on rectified images is often referred to as *disparity*.

The block matching algorithm described above can be used to estimate the disparity in rectified stereo pairs by restricting it to consider only horizontal offsets (see Figure 10). However, for estimating the depth it is useful to compute disparities with subpixel precision. Algorithm 4 extends the block matching for integer offsets to consider subpixel disparities. The principle is to compute several disparity maps with integer disparities, where in each pair one image is shifted by a subpixel quantity, and then combine the results in a single subpixel disparity map. To attain a precision of $1/n$-th of pixel for both the direct and reverse disparity maps, the integer precision algorithm must be invoked $2n$ times, meaning that the overall cost depends linearly on the precision $O(NnM)$ (where N is the number of pixels and M the number of offsets). The same subpixel procedure can be applied for estimating subpixel bi-dimensional offsets. However, the cost becomes $O(Nn^2M)$, which dims the value of the integral image with respect to other subpixel techniques for *optical flow* computation [14, 19].

3.4 Comparison with Exhaustive Block Matching

As hinted in the introduction the performance of the exhaustive block matching decreases when the patch size grows, whereas the performance of integral images does not significantly depend on patch size. The results summarized in Table 1 corroborate this. We also note that for patches smaller than 3×3 the exhaustive search becomes more efficient than applying the integral image.

[5] http://dx.doi.org/10.5201/ipol.2014.57

Algorithm 4: Subpixel stereo block matching using integral image.

input: Images: u and v. A subpixel precision $s < 1$. A disparity range: *offsets*.
output: Disparities and minimum matching costs: matchOffset, minCost.
Reverse matches: R_matchOffset, R_minCost.

```
/* initialize the costs                                                    */
1  minCost ← ∞ ;
2  R_minCost ← ∞ ;
/* subpixel block matching:  u to v                                        */
3  for shift ∈ {ns : n ∈ ℕ, ns < 1} do
4      shifted_u ← Shift_Horizontally(u, shift);
5      [newOff,newCost,R_newOff,R_newCost] ← BlockMatching(shifted_u, v, offsets);
6      for p ∈ u do
7          if minCost(p) > newCost(p) then
8              minCost(p) ← newCost(p);
9              matchOffset(p) ← newOff(p) + shift;
/* subpixel block matching:  v to u                                        */
10 for shift ∈ {ns : n ∈ ℕ, ns < 1} do
11     shifted_v ← Shift_Horizontally(v, shift);
12     [newOff,newCost,R_newOff,R_newCost] ← BlockMatching(u, shifted_v, offsets);
13     for p ∈ v do
14         if R_minCost(p) > R_newCost(p) then
15             R_minCost(p) ← R_newCost(p);
16             R_matchOffset(p) ← R_newOff(p) + shift;
```

Patch size (side r)	Integral image	Exhaustive search
3	6 s	6 s
5	6 s	15 s
7	6 s	28 s
9	6 s	48 s
11	6 s	71 s

Table 1: Performance comparison of block matching with integral image against the exhaustive search. The former is $O(MN)$ while the latter is $O(r^2MN)$, where M is the number of considered offsets, N is the number of pixels in the image and the side of the patch is r. Here $N = 512^2$ and $M = 100$. The execution times are obtained running a C implementation of the algorithms on a 1.8Ghz CPU.

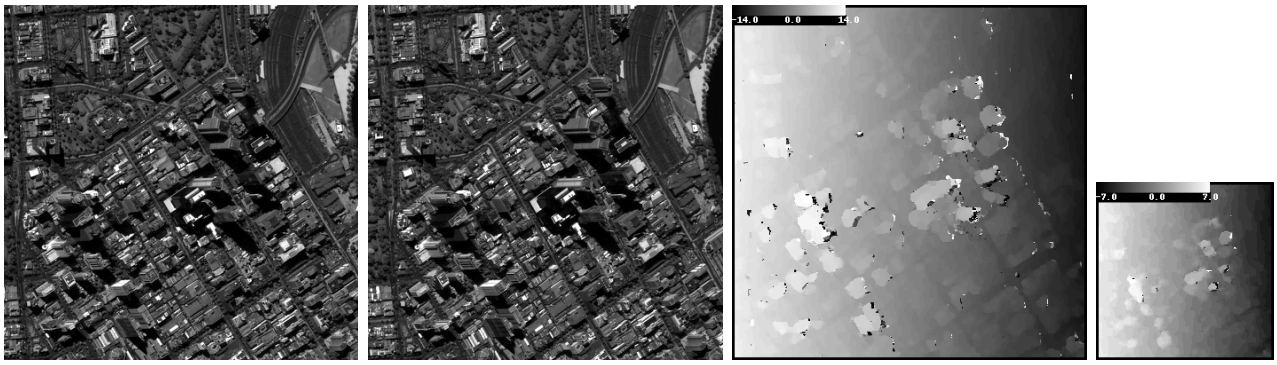


Figure 11: **Disparity range.** On the right we show the disparity corresponding to the stereo pair on the left. The range of the offset spans a large set of disparities because of the "ground plane". This tendency could be determined at lower resolutions at a lower cost.

There are however many cases in which an exhaustive block matching is preferable to using the integral images.

- **Non overlapping patches.** A common trick to accelerate block matching is to compute the correspondences only on a coarse grid over the reference image. However, there is no benefit in using the integral image to compute the costs if the considered patches do not overlap. Indeed, when computing the correspondences for points on a grid with a stride bigger than the patch width, there is no gain in using the integral image, since the considered patches are all disjoint.

- **Matching costs based on feature vectors.** The block matching can be performed by comparing vectors of local features extracted from the image, instead of comparing the pixels in a patch. These features vectors are high level descriptions of the patch which usually introduce some invariance.

  However, as in the previous case, there is no benefit in using the integral image if the patch representations do not overlap. The integral image may be useful, however, for computing the features themselves [1].

- **Matching costs using non uniform weights.** A drawback of block matching methods is their inability to distinguish distinct heights within the patch. This leads to incorrect estimations near the objects' boundaries and to the appearance of artifacts like the *foreground fattening*. The *adaptive weights* technique [27] prevents some of these issues by introducing weights in the computation of the matching costs. The weights are computed taking into account the image geometry in order to avoid considering more than one object in each patch. This leads to improved results near the boundaries of the objects.

  Although the weighted matching costs can be computed using integral images (see Section 2.2), their cost will be proportional to the complexity of the weight function (for weights $w(x, y)$ it is proportional to the cardinality of the set $\{(x, y) : \frac{d^2}{dx^2} w(x, y) \neq 0\}$). Usually the adaptive weights are incorporated as a refinement step operating over an initial coarse disparity map.

- **Variable offset ranges.** The block matching computational cost depends linearly on the number of offsets being evaluated; for this reason it is always preferable to explore a small range of offsets. In many cases, although the overall offset range can be large, the range remains locally small. The stereo pair shown in Figure 11 illustrates this situation. In spite of the fact that the disparities induced by the structures are all relatively small, the disparity

map is dominated by a global motion due to the change of view angle, which could be easily estimated at a lower resolution. This idea of progressively refining coarse approximations of the offset field is at the base of coarse-to-fine [20] techniques for block matching.

An exhaustive block matching algorithm can be easily adapted to implement a coarse-to-fine refinement, just by locally adapting the offset range. But, an algorithm based on integral images (such as Algorithm 3) must test every offset for all the pixels, which is inefficient for a coarse-to-fine technique. A possible way to adapt the disparity range in Algorithm 3 consists in splitting the reference image in small overlapping tiles, and then applying the algorithm on each tile with a more reduced range.

3.5 Comparison with Fourier Methods

The convolution theorem provides another way to accelerate the block matching. Any matching cost that can be expressed in terms of $\mathrm{Prod}(\mathbf{p}, \mathbf{q})$ (including SSD) admits a re-writing as convolution of images

$$\mathrm{Prod}(\mathbf{p}, \mathbf{x}) = (\tilde{u} * \tilde{v})(\mathbf{x}),$$

where $\tilde{u}(\mathbf{x}) = \mathbf{1}_{B(0,r)}(-\mathbf{p}-\mathbf{x})u(-\mathbf{p}-\mathbf{x})$ and $\tilde{v}(\mathbf{x}) = v(\mathbf{x})$. Applying the convolution theorem we have

$$\mathrm{Prod}(\mathbf{p}, \mathbf{x}) = (\tilde{u} * \tilde{v})(\mathbf{x}) = \mathcal{F}^{-1}\{\mathcal{F}\{\tilde{u}\} \cdot \mathcal{F}\{\tilde{v}\}\}(\mathbf{x}).$$

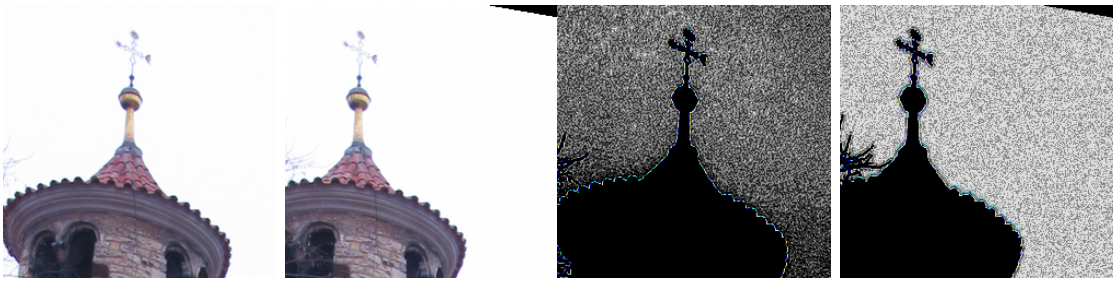
This implies that, for a patch of size $r \times r$ of the reference image (centered at $\mathbf{p}$), all the matching costs within the search window can be computed at a cost proportional to the Fourier transform of the search region. Then, it is the size of the search region that determines the computational cost. Indeed, assuming that the M offsets to be evaluated are arranged in a squared region, and that $r^2 < M$, then the cost becomes approximately $O(M \log_2(M))$ (with a conservative constant of 30 due to the FFTs). We deduce that the Fourier transform computes the matching costs faster than the exhaustive search, as soon as $r^2 > 30 \log_2(M)$. This is the case for template matching [17], where the patches (r^2) and the search regions (M) are both large.

A key characteristic of template matching is that the set of features to be matched is usually small and sparse, be they points of interest or tracking points. In this case where there is no overlap between the features, the integral image will perform as the exhaustive search with $O(Mr^2)$ operations for each feature, while the Fourier method's cost is $O(M \log_2(M))$ operations per feature. Conversely, applying the Fourier method for computing the matches for all the pixels in the reference image is $O(NM \log_2(M))$, which is worse than using the integral image.

4 Limitations

In the previous sections we pointed out some limitations of using integral images for block matching, let us summarize them here:

- The **limited performance gain** with respect to exhaustive block matching for small patch sizes.
- There is **no performance gain** with respect to exhaustive block matching when computing the integral image on a coarse grid with non overlapping patches.
- Integral images **cannot be used to implement all matching costs**. For example, they do not adapt to ZSAD. Neither are they applicable to **matching costs based on feature vectors** or to computing costs with **adaptive weights**.



(a) stereo pair (b) increased contrast

(c) exhaustive BM (d) integral image BM (e) bands

Figure 12: **Ghost artifacts in block matching.** Subfigure (a) shows a crop from a stereo pair. Although the sky looks saturated it is not uniform. To highlight the noisy values in the sky region the pair is visualized in (b) with higher contrast. The disparity map obtained using exhaustive block matching is shown in (c). As expected, the matches in the sky are erratic. The disparity map shown in (d) is computed using a single precision integral image. Note the appearance of a "ghost" of the main structure in the sky region, caused by the truncation errors in the integral image representation. The result obtained using a double precision integral image (not shown) coincides with the one of the exhaustive matching. The result obtained with a single precision integral image but processing the image by horizontal bands is shown in (e). Although the artifact has disappeared the result differs from the exact one, which indicates that some truncation errors are still present.

- The limited **precision of the data type used for representing the integral image** was remarked in Section 2.3. Let us illustrate how this issue affects the performance of block matching. Figure 12 shows a disparity map with a ghost artifact which is the product of the insufficient precision of the integral image representation. Indeed, the single precision float is insufficient in this case. There are two ways to deal with this issue, one by reducing the size of the integral image, and the other by involving a higher precision. A possibility to reduce the size of the integral image is to process the input images by bands. Figure 12 illustrates the results of both approaches.

5 Take Home Message

The integral image is a powerful tool for computer vision, that is also effective for accelerating block matching algorithms, as long as the matching cost can be expressed as a sum of pixel-wise costs and the offset set to be tested is small. Its performance comes from a re-arrangement of the computations that capitalizes on patch overlap. Therefore, if the "query patches" do not overlap, it is more convenient to resort to Fourier based correlation, as it is the case for template matching. Moreover, for applications where the patches are encoded by feature vectors, there is no advantage

in using the integral image. Lastly, single precision should be avoided for integral images, as it may cause truncation errors.

A Comparison of Different Implementations of the Integral Image Loop

As shown in previous sections of this article, the integral image can be used as a building-block for many computations, so its performance is critical and optimization efforts focusing on this core loop could benefit other algorithms.

We present in this appendix a comparison of three versions of the integral image computation algorithm. These three versions, detailed hereafter, only differ by the order of the operations and storage of the temporary values, all implemented in general-purpose C language.

Two Loops

We can compute the integral image by performing cumulative sums by row and column in two passes over the source image (Algorithm 5). This version avoids the use of temporary variables, but it implies more memory traffic with the two passes.

Vector Accumulator

This version interprets the Viola-Jones [24] formula

$$\begin{aligned} s(x,y) &= s(x,y-1) + i(x,y) \\ ii(x,y) &= ii(x-1,y) + s(x,y), \end{aligned}$$

as a sum over columns. With arrays stored by row (C-style), this implies the use of a vector accumulator to store the current column-wise sums (Algorithm 6).

Scalar Accumulator

Since the Viola-Jones formula is symmetric, one can swap row and column indices and summing over rows leads to another variant with a single scalar accumulator to store the current row-wise sum (Algorithm 7).

We implemented these three variants in C and compared their speed on arrays of pseudo-random values in $[0, 1]$ stored as long double variables. Processing time for these core routines only was measured from a single thread execution with high-precision POSIX clock_*() calls, on an idle Linux 3.10 system with two CPU cores (Intel Core i5 2.4GHz), no hyper-threading, and no CPU frequency scaling. For each array size, 32 measures were collected and their median was kept, hence discarding outliers.

The result, displayed in Figure 13 and Table 2, shows that the "scalar accumulator" version is faster for all array sizes (2^{12} to 2^{24} pixels), all compilers (gcc 4.7.3 and clang 3.2) and all optimization settings (-O0 and -O3). On average over the input size, the "scalar accumulator" version is 1.3 to 2 times faster than the "two loops" variant, depending on the compiler and compilation options. Similar results were obtained with different optimization settings, other compilers, and other hardware architectures[6].

[6] Additional tests were performed with -O1 and -O2 optimization settings on gcc and clang, with the icc 12.0.4, suncc 5.11, and tcc 0.9.25 compilers, and with the gcc compiler on Intel Xeon X7560 and Marvell ARM Feroceon 88FR131 CPUs.

Algorithm 5: Two loops algorithm variant.

input: An image of size $nx \times ny$: i
output: The integral image of size $nx \times ny$: ii

```
1 for x ← 0 to nx − 1 do
      /* copy the first row                                   */
2     ii(x, 0) ← i(x, 0);
3 for y ← 1 to ny − 1 do
      /* cumulative sum over y indices                        */
4     for x ← 0 to nx − 1 do
5         ii(x, y) ← i(x, y) + ii(x, y − 1);
6 for y ← 0 to ny − 1 do
      /* cumulative sum over x indices                        */
7     for x ← 1 to nx − 1 do
8         ii(x, y) ← ii(x, y) + ii(x − 1, y);
```

Algorithm 6: Vector accumulator algorithm variant.

input: An image of size $nx \times ny$: i
output: The integral image of size $nx \times ny$: ii

```
1 s(0 . . . nx − 1) ← 0;        /* vector accumulator of length nx initialized to 0 */
2 for y ← 0 to ny − 1 do
3     s(0) ← s(0) + i(0, y);
4     ii(0, y) ← 0 + s(0);
5     for x ← 1 to nx − 1 do
6         s(x) ← s(x) + i(x, y);
7         ii(x, y) ← ii(x − 1, y) + s(x);
```

Algorithm 7: Scalar accumulator algorithm variant.

input: An image of size $nx \times ny$: i
output: The integral image of size $nx \times ny$: ii

```
1 ii(0, 0) ← i(0, 0);
2 for x ← 1 to nx − 1 do
3     ii(x, 0) ← ii(x − 1, 0) + i(x, 0);
4 for y ← 1 to ny − 1 do
5     s ← i(0, y);          /* scalar accumulator                */
6     ii(0, y) ← ii(0, y − 1) + s;
7     for x ← 1 to nx − 1 do
8         s ← s + i(x, y);
9         ii(x, y) ← ii(x, y − 1) + s;
```

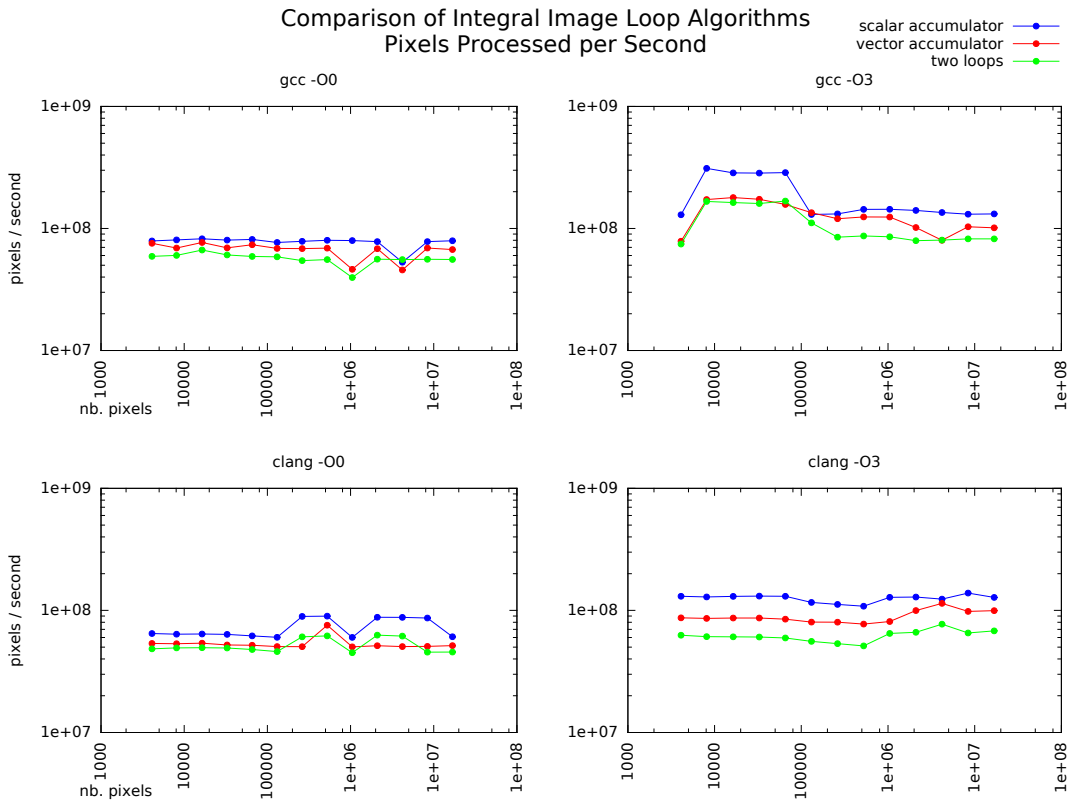


Figure 13: Comparison of three implementations of the core loop. The "scalar accumulator" version is faster on different compilers and optimized compilation settings. These measures, on arrays filled with 2^{12} to 2^{24} pseudo-random values in $[0, 1]$ stored as long double variables, are the median of 32 independent runs. Average and normalized performance are summarized in Table 2. The test platform has an Intel Core i5 CPU at 2.4GHz (frequency scaling disabled), with 2 cores, no hyper-threading, and the software was run on a single thread, on an idle system, without memory paging to disk (swapping). Compilers here are gcc 4.7.3 and clang 3.2, on a Linux 3.10 system.

	scalar	vector	two loops
gcc -O0	7.74218e+07	6.67072e+07	5.66965e+07
	1.3655	*1.1765*	*1*
gcc -O3	1.83359e+08	1.27013e+08	1.09579e+08
	1.6733	*1.1591*	*1*
clang -O0	7.23631e+07	5.35246e+07	5.18055e+07
	1.3968	*1.0332*	*1*
clang -O3	1.25750e+08	8.93577e+07	6.20220e+07
	2.0275	*1.4407*	*1*

Table 2: Average performance of three implementations of the core loop, in pixels per second. This measure is the average of the values presented in Figure 13, i.e. on square arrays of 2^{12} to 2^{24} values. Numbers in italic are normalized versions of the previous lines, for easier comparison.

We propose two explanations for the difference. First, the "two loops" variant requires two passes over the whole array, which implies more time to fetch the data from RAM to CPU and back when the array doesn't fit entirely on CPU cache memory. This might explain why Viola and Jones mentioned that with their formula "the integral image can be computed *in one pass* over the original image". Then, using a "scalar accumulator" instead of a "vector accumulator" means this variable can live entirely as a single CPU register; it doesn't need any space on the CPU cache memory, hence less CPU cycles to load cache values into registers and back, and less cache miss.

This comparison illustrates how much implementation details can matter and make a difference between two algorithm variants with the same mathematical properties and same arithmetic operations. When even small performance improvements are important, as in core routines of generic algorithmic building blocks, anything less than the source code doesn't really tell us how the software is going to behave[7].

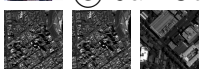
We intend to complete this analysis with more subtle variants in a future work focused on implementation details, such as the choice of the variable data type, floating-point precision control, loop unrolling, vector instructions, parallel processing, and other optimizations specific to a compiler, architecture or programming language.

Acknowledgments

The authors thank to the anonymous reviewers for their comments and suggestions, which have notably contributed to the improvement of our work. This work was supported in part by the Centre National d'Études Spatiales (MISS Project) and the European Research Council (advanced grant "Twelve Labours").

Image Credits


© Jan Čech [4], crop from StMartin rectified stereo pair[8]


© CNES, satellite image of Melbourne 2012.

References

[1] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, *Speeded-Up robust features (SURF)*, Computer Vision and Image Understanding, 110 (2008), pp. 346–359. http://dx.doi.org/10.1016/j.cviu.2007.09.014.

[2] S. Birchfield and C. Tomasi, *A pixel dissimilarity measure that is insensitive to image sampling*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 20 (1998), pp. 401–406. http://dx.doi.org/10.1109/34.677269.

[3] A. Buades, B. Coll, and J. M. Morel, *A review of image denoising algorithms, with a new one*, SIAM Multiscale Modeling & Simulation, 4 (2005), pp. 490–530. http://dx.doi.org/10.1137/040616024.

[7] The last part of this sentence is based on a quote of Douglas Crockford in *Coders at Work: Reflections on the Craft of Programming*, by Peter Seibel.

[8] http://cmp.felk.cvut.cz/~cechj/GCS/stereo-images/StMartin/

[4] J. Cech and R. Sara, *Efficient Sampling of Disparity Space for Fast And Accurate Matching*, in Proceedings of the 2007 Conference on Computer Vision and Pattern Recognition, IEEE, June 2007, pp. 1–8. http://dx.doi.org/10.1109/CVPR.2007.383355.

[5] F.C. Crow, *Summed-area tables for texture mapping*, ACM SIGGRAPH Computer Graphics, 18 (1984), pp. 207–212. http://dx.doi.org/10.1145/964965.808600.

[6] J. Delon and A. Desolneux, *Stabilization of Flicker-like effects in image sequences through local contrast correction*, SIAM Journal on Imaging Sciences, 3 (2010), pp. 703–734. http://dx.doi.org/10.1137/090766371.

[7] K.G. Derpanis, E.T.H. Leung, and M. Sizintsev, *Fast scale-space feature representations by generalized integral images*, in Proceedings of the 2007 International Conference on Image Processing, vol. 4, IEEE, Oct. 2007, pp. 521–524. http://dx.doi.org/10.1109/ICIP.2007.4380069.

[8] O. Faugeras, T. Viéville, E. Theron, J. Vuillemin, B. Hotz, Z. Zhang, L. Moll, P. Bertin, H. Mathieu, P. Fua, G. Berry, and C. Proy, *Real-time correlation-based stereo: algorithm, implementations and applications*, Rapport de recherche RR-2013, INRIA, 1993. http://hal.inria.fr/inria-00074658/PDF/RR-2013.pdf.

[9] L.A. Ferrari, P.V. Sankar, J. Sklansky, and S. Leeman, *Efficient two-dimensional filters using B-spline functions*, Computer Vision, Graphics, and Image Processing, 35 (1986), pp. 152–169. http://dx.doi.org/10.1016/0734-189X(86)90024-1.

[10] L.A. Ferrari and J. Sklansky, *A fast recursive algorithm for binary-valued two-dimensional filters*, Computer Vision, Graphics, and Image Processing, 26 (1984), pp. 292–302. http://dx.doi.org/10.1016/0734-189X(84)90214-7.

[11] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, Cambridge University Press, 2nd ed., Apr. 2004. ISBN 0521540518.

[12] P.S. Heckbert, *Filtering by repeated integration*, ACM SIGGRAPH Computer Graphics, 20 (1986), pp. 315–321. http://dx.doi.org/10.1145/15886.15921.

[13] H. Hirschmuller and D. Scharstein, *Evaluation of stereo matching costs on images with radiometric differences*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 31 (2009), pp. 1582–1599. http://dx.doi.org/10.1109/TPAMI.2008.221.

[14] B.K.P. Horn and B.G. Schunck, *Determining optical flow*, Artificial Intelligence, 17 (1981), pp. 185–203. http://dx.doi.org/10.1016/0004-3702(81)90024-2.

[15] J. Jain and A. Jain, *Displacement measurement and its application in interframe image coding*, IEEE Transactions on Communications, 29 (1981), pp. 1799–1808. http://dx.doi.org/10.1109/TCOM.1981.1094950.

[16] B.W. Kernighan and D.M. Ritchie, *The C programming language*, Prentice-Hall, 2nd ed., 1988. ISBN 0131103628.

[17] J.P. Lewis, *Fast template matching*, in Vision Interface 95, Canadian Image Processing and Pattern Recognition Society, May 1995, pp. 120–123. http://www.cipprs.org/papers/VI/VI1995/pp120-123-Lewis-1995.pdf.

[18] R. Lienhart and J. Maydt, *An extended set of Haar-like features for rapid object detection*, in Proceedings of the 2002 International Conference on Image Processing, vol. 1, IEEE, Sep. 2002, pp. 900–903. http://dx.doi.org/10.1109/ICIP.2002.1038171.

[19] B.D. Lucas and T. Kanade, *An iterative image registration technique with an application to stereo vision*, in Proceedings of the 7th International Joint Conference on Artificial Intelligence, vol. 2, Morgan Kaufmann Publishers Inc., Aug. 1981, pp. 674–679. http://ijcai.org/Past%20Proceedings/IJCAI-81-VOL-2/PDF/017.pdf.

[20] L.H. Quam, *Hierarchical warp stereo*, in Readings in computer vision: issues, problems, principles, and paradigms, Martin A. Fischler and Oscar Firschein, eds., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987, pp. 80–86. http://www.dtic.mil/dtic/tr/fulltext/u2/a461877.pdf.

[21] D. Scharstein and R. Szeliski, *A taxonomy and evaluation of dense two-frame stereo correspondence algorithms*, International Journal of Computer Vision, 47 (2002), pp. 7–42. http://dx.doi.org/10.1023/A:1014573219977.

[22] P.Y. Simard, L. Bottou, P. Haffner, and Y. LeCun, *Boxlets: a fast convolution algorithm for signal processing and neural networks*, in Proceedings of the 1998 Conference on Advances in Neural Information Processing Systems 11, MIT Press, Nov. 1998, pp. 571–577. http://books.nips.cc/papers/files/nips11/0571.pdf.

[23] O. Veksler, *Fast variable window for stereo correspondence using integral images*, in Proceedings of the 2003 Conference on Computer Vision and Pattern Recognition, vol. 1, IEEE, June 2003, pp. 556–561. http://dx.doi.org/10.1109/CVPR.2003.1211403.

[24] P. Viola and M.J. Jones, *Robust real-time face detection*, International Journal of Computer Vision, 57 (2004), pp. 137–154. http://dx.doi.org/10.1023/B:VISI.0000013087.49260.fb.

[25] J. Wang, Y. Guo, Y. Ying, Y. Liu, and Q. Peng, *Fast non-local algorithm for image denoising*, in Proceedings of the 2006 International Conference on Image Processing, IEEE, Oct. 2006, pp. 1429–1432. http://dx.doi.org/10.1109/ICIP.2006.312698.

[26] L. Williams, *Pyramidal parametrics*, ACM SIGGRAPH Computer Graphics, 17 (1983), pp. 1–11. http://dx.doi.org/10.1145/964967.801126.

[27] K.J. Yoon and I.S. Kweon, *Adaptive support-weight approach for correspondence search*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 28 (2006), pp. 650–656. http://dx.doi.org/10.1109/TPAMI.2006.70.