VIPS: An image processing system for large images

John Cupitt[†], Kirk Martinez[‡]

[†]The National Gallery, Trafalgar Square, London WC2N 5DN
[‡]Birkbeck College, 43 Gordon Square, London WC1H 0PD

Abstract

This paper describes VIPS (VASARI Image Processing System), an image processing system developed by the authors in the course of the EU-funded projects VASARI[1] (1989–1992) and MARC[2] (1992–1995). VIPS implements a fully demand-driven dataflow image IO (input-output) system. Evaluation of library functions is delayed for as long as possible. When evaluation does occur, all delayed operations evaluate together in a pipeline, requiring no space for storing intermediate images and no unnecessary disc IO. If more than one CPU is available, then VIPS operations will automatically evaluate in parallel, giving an approximately linear speed-up.

The evaluation system can be controlled by the application programmer. We have implemented a user-interface for the VIPS library which uses expose events in an X window rather than disc output to drive evaluation. This makes it possible, for example, for the user to rotate an 800 MByte image by 12 degrees and immediately scroll around the result.

Keywords: Dataflow, demand-driven, image processing, threading, large images, lazy evaluation.

This paper originally appeared as

J. Cupitt and K. Martinez, *VIPS: an image processing system for large images*, Proc. SPIE Vol. 2663, p. 19 – 28, 1996.

1 INTRODUCTION

The VASARI project[1] developed a colorimetric scanner capable of making images directly from paintings. These images are used for assessing change in surface appearance and therefore need to be of very high resolution: up to 20 pels per millimetre. For a 1m by 1m painting, a single image can therefore reach 1.6GBytes. We were unable to find a commercial image processing package which could efficiently handle images of this size and so resolved to write our own.

VIPS implements a fully demand-driven image IO system. The API (Application Programmer's Interface) is very like a conventional image processing package: the programmer opens the input images, calls a number of processing functions and writes the final output image back to disc again. However, behind the scenes, VIPS delays most computation until the final write. When the final image processing function is called, all of the operations evaluate together, sucking the source images through the pipeline of operations in small pieces. In contrast, when more conventional image processing systems evaluate a series of operations, each of one has to evaluate completely before the next can start. This has the result that as the images being processed become larger, more and more disc space has to be found to store intermediate images and more and more time has to be given over to disc IO. A series of VIPS operations will meld together to become, in effect, a single operation, requiring no disc space for intermediates and no unnecessary disc IO.

VIPS operations have other advantages: if more than one CPU is available, then operations will automatically evaluate in parallel giving (depending upon the complexity of the operations) an approximately linear speed-up. The evaluation system can be controlled by the application programmer: we have implemented a user interface for the VIPS library which uses expose events in an X window rather than disc output to drive evaluation. This makes it possible for the user to (for example) rotate an 800 MByte image by 12 degrees and immediately view the result (or at least as much of the rotated image as will fit on the screen at any one time).

2 PARTIAL IMAGE DESCRIPTORS

This section introduces the application programmer's view of VIPS. The key idea here is the partial image descriptor. Image descriptors are commonly used in image processing systems to represent an input or output image — in VIPS, partial image descriptors are used by the application programmer to control lazy evaluation. When a VIPS image processing function writes to a partial descriptor, it sets up the descriptor (setting fields for size, type and so on), but delays actually writing any image data. Instead, it attaches a set of pointers to functions which subsequent image processing operations can use to generate any part of the image they wish.

Consider this fragment of code to add a constant to an image and force the result to unsigned 8 bit:

```
#include <vips.h>

int
add_const( IMAGE *in, int offset, IMAGE *out )
{
```

Open an image descriptor we can use as an intermediate. `im_open_local()` opens a new descriptor local to an existing descriptor. When `out` is closed (the responsibility of our caller) `t1` will also be automatically closed. The final `"p"` is the mode — `"p"` means "open a partial image descriptor".

```
    IMAGE *t1 = im_open_local( out, "add_const:1", "p" );
```

Check the open was OK and add the constant. `im_lintra( a, in, b, out )` (for linear transform) calculates `out = a*in + b`.

```
    if( !t1 || im_lintra( 1.0, in, offset, t1 ) )
        return( -1 );
```

`im_lintra()` always writes a float image unless the input image is double, complex or double complex, in which case it writes the same type as the input. In any event, we need for force the output of `im_lintra()` back to unsigned char.

```
    if( im_clip2uc( t1, out ) )
        return( -1 );
```

Success! We can simply return 0.

```
    return( 0 );
}
```

The API is, in other words, very much like that of a conventional image processing library. This function might be called from an application with:

```
int
main( int argc, char **argv )
{
    IMAGE *in *out;
    int off;
    error_exit() is a convenience function provided by VIPS — it prints
    a diagnostic message to stderr and causes the program to exit with error
    status.

    if( argc != 4 )
        error_exit( "usage: %s in num-const out", argv[0] );

    in = im_open( argv[1], "r" );
    off = atoi( argv[2] );
    out = im_open( argv[3], "w" );
    if( !in || !out )
        error_exit( "unable to open images" );

    if( add_const( in, off, out ) )
        error_exit( "error adding const" );

    if( im_close( in ) || im_close( out ) )
        error_exit( "error closing images" );

    return( 0 );
}
```

Now consider the evaluation of this program. `im_lintra()` is writing to a partial descriptor, so rather than evaluating immediately, it will just set up `t1` with the appropriate sizes and types, and return immediately. The second function to be called, `im_clip2uc()`, will write to a descriptor attached to a disc file. VIPS will spot this and start evaluating.

The evaluation system decides how many threads should be used to evaluate this pipeline and allocates a buffer for each. The size and shape of the buffers is determined by the sorts of operations in the pipeline (see section 3.1 for an explanation of this mechanism). In this example, the buffers will be one pel high, and as wide as the image. The evaluator now starts a thread to fill each of the buffers it allocated. When a thread finishes, the buffer of pels it has created are written to disc and the thread restarted on a new area of image. VIPS includes mechanisms which image processing functions can use to control thread starting and stopping, allowing functions like `im_clip2uc()` to produce reliable overflow counts even when they are threaded — see section 3 below.

As `im_clip2uc()` receives each demand for image data it will in turn demand image data from its input. These demands will pass to `im_lintra()` which will, again in turn, demand data from the input file. When the first strip of data arrives from the disc, `im_lintra()` will process it to produce a strip of float pels which it will return to `im_clip2uc()`. `im_clip2uc()` will then process the line of float pels to produce a line of unsigned char pels, which it will in turn return to the VIPS evaluator to be written to disc. This process is shown diagrammatically in figure 1.

3 INSIDE A VIPS OPERATION

This section briefly explains the mechanisms used inside VIPS operations to support threading and demand-driven evaluation. It might seem that an image processing operation written in this way would be significantly more complicated than one written to a more traditional API — as we hope to show here, this is not the case.

There are two principal ideas: first, the *region*. This is a sub-area of an image, with a small amount of local memory. The core of each image processing operation is a generate function which, when given a region, will fill it with image data.

The second idea is the *sequence*. This is the representation of parallelism inside an operation. Image processing operations are split into four parts: a main function which provides the public interface to the operation, the generate function described above, and start and stop functions. Start functions perform

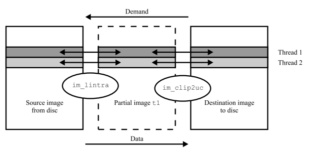


Figure 1: Evaluation of `add_const()`

any per-thread initialise operations (such as allocating input regions, or local scratch memory), and produce a sequence value. This sequence value is passed in to all subsequent generate functions for this thread. Stop functions are called to destroy the sequence value once evaluation has finished. The start and stop functions for an operation are guaranteed by VIPS to be mutually exclusive. This makes it possible for threads to cooperatively produce statistics, such as overflow counts. See figure 2.

Here is the generate function for an operation which finds the photographic negative of an unsigned 8-bit image.

```
static int
invert_gen( REGION *or, REGION *ir, IMAGE *in )
{
```

`or` is the output region which this call to `invert_gen()` has to fill with image data. `or->valid` holds the coordinates for the top, left, bottom and right of the area to be made. `ir` is the sequence value for this generate function, and in this simple case, it is just a region defined on the input image.

```
Rect *r = &or->valid;
int x, y, b;
```

The first step is to demand the part of the input image which will be needed to generate `or->valid`.

```
if( im_prepare( ir, r ) )
    return( -1 );
```

Now `invert_gen()` just has to loop over the output area, reading the corresponding input pel, and writing the inverted value. `addr()` is a macro which, given a region and an (x, y) coordinate, returns a pointer to the corresponding pel.

```
for( y = r->top; y < bottom( r ); y++ ) {
    unsigned char *p = (unsigned char *) addr( ir, r->left, y );
    unsigned char *q = (unsigned char *) addr( or, r->left, y );
```

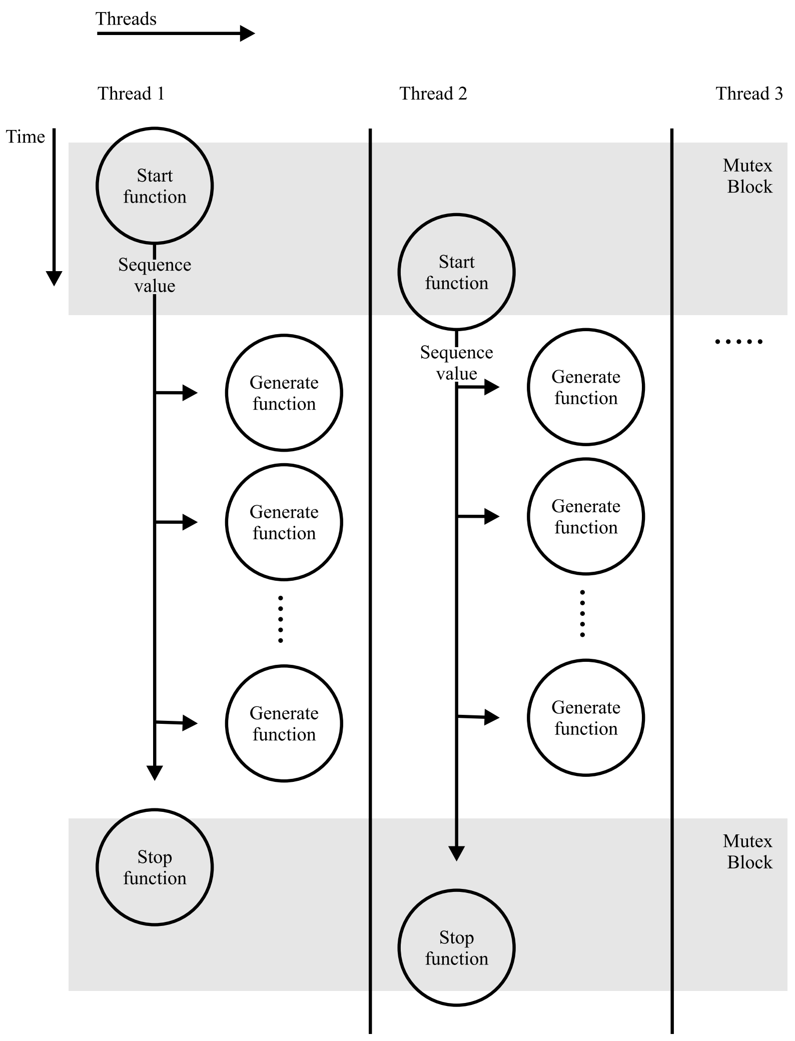


Figure 2: Sequences and threads inside an operation

```
            for( x = r->left; x < right( r ); x++ )
                for( b = 0; b < in->Bands; b++ )
                    *q++ = 255 - *p++;
        }

        return( 0 );
    }
```

And finally, here is the main function for the invert operation. This is what applications (such as the `main()` program in section 2 earlier) might call.

```
int
invert( IMAGE *in, IMAGE *out )
{
```

First, check the parameters. `im_piocheck()` checks that partial input and output is sensible between the two descriptors. The `if` statement just checks that the image is unsigned 8-bit.

```
    if( im_piocheck( in, out ) )
        return( -1 );
    if( in->Coding != NOCODING || in->Coding != FMTUCHAR ) {
        im_errormsg( "invert: uncoded uchar only" );
        return( -1 );
    }
```

`im_cp_desc()` copies attributes from the input descriptor to the output. For this simple operation, the output image has the same size and type as the input, so no changes are necessary.

```
    if( im_cp_desc( out, in ) )
        return( -1 );
```

`im_demand_hint()` hints to the VIPS evaluator what kind of coordinate transformation this operation performs. The evaluator uses the set of hints set by operations in a pipeline to select a shape for the areas of pels it demands from the end of the pipeline. See section 3.1 below.

```
    if( im_demand_hint( out, IM_THINSTRIP, in, NULL ) )
        return( -1 );
```

Finally, `im_generate()` runs the operation. The parameters are the output image, the start function, the generate function, a stop function, and any extra parameters that are needed by the generate function. This example uses the default start and stop functions supplied by VIPS. These can be overridden if required.

```
    if( im_generate( out,
        im_start_one, invert_gen, im_stop_one, in ) )
        return( -1 );

    return( 0 );
}
```

3.1 Demand hints

One of the most significant factors affecting processing speed when images become large is the operation's paging behaviour. Disc drives are fastest when doing long sequential reads, and slowest when they are forced to do many small, scattered reads. If an operation causes excessive seeking on the input images, it can run very slowly. For example, consider a 90° rotate operation on a large image. If it asked to generate the output image in a series of strips, each one pel high and as wide as the output, then it will have to scan the whole of the input image for each output line.

The solution VIPS adopts is to let each operation hint to the evaluator what kind of demand geometry it prefers. Before starting to evaluate a pipeline, VIPS examines the hints each operation has set, and chooses the lowest common denominator. The three types of hint which operations can set, in order of increasing power, are:

`THINSTRIP` This is the hint set by `invert()` in section 3 above. It means strips as wide as the output image and one pel high. It is suitable for operations which do not transformation coordinates.

`FATSTRIP` This hint is for area operations such as convolution or morphology. Output is demanded in strips as wide as the output image and up to about 50 pels high.

`SMALLTILE` This is the geometry set by operations which radically transform coordinates, such as the 90° rotate above. Output is demanded in small tiles, about 50 pels square.

3.2 Other features

There are several other features of the VIPS image IO system which are beyond the scope of this paper. The `im_generate()` call above creates a data source — there is an analogous function called `im_iterate()` which creates a data sink. It is used by operations which consume images but which do not produce image output, such as an operation to find the average pel value in an image. The same call is used to calculate images for display in the GUI (Graphical User-Interface) we have developed.

The regions used in the example code above all had local memory. Regions can instead be made to refer to pels in another image, or even in another region. This makes it possible to write operations which elide — for example, VIPS has an operation called `im_extract()` which extracts an area from an image, forming a smaller image. If `im_extract()` is composed with another operation, it is able to work by copying pointers rather than by copying image data, effectively vanishing. In this way, VIPS is able to provide region-of-interest behaviour for all operations at no extra cost to the programmer.

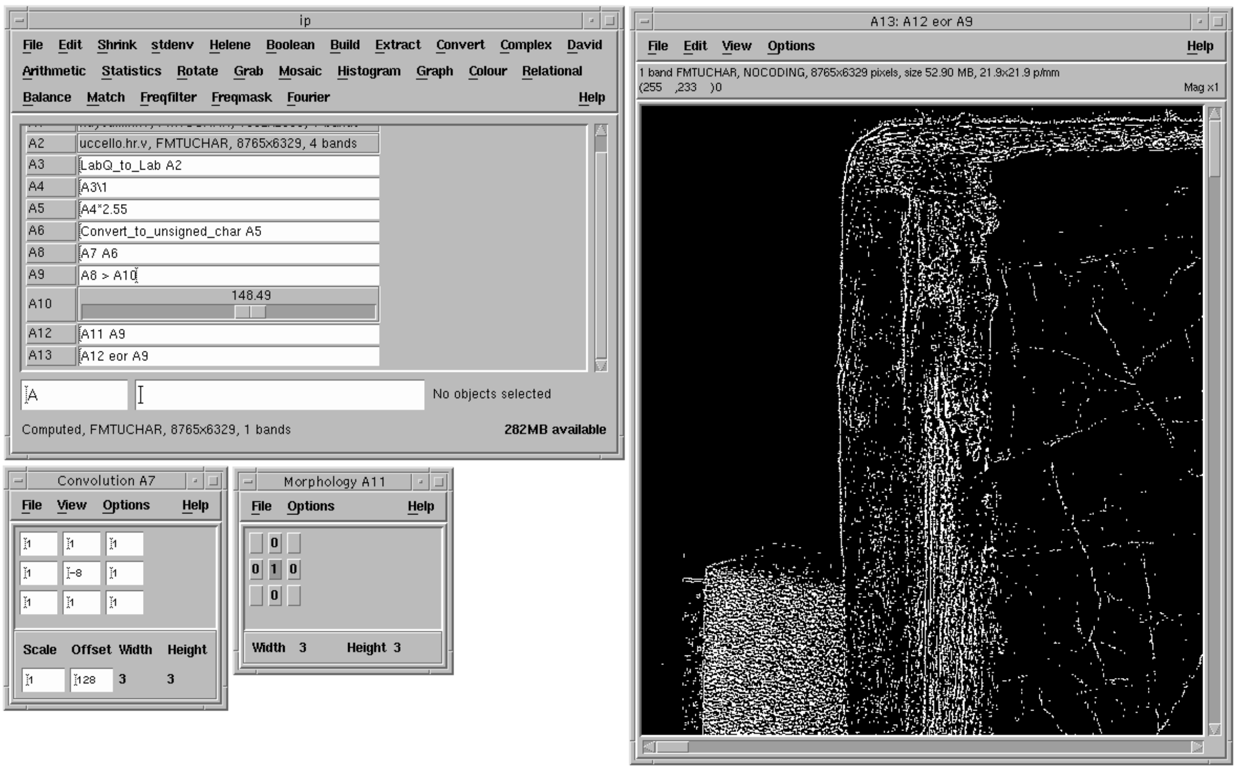
VIPS includes an extensible function dispatch system. This is a simple database which sits between user-interfaces and the library. Programmers may add new image processing operations (and even new object types) to the database and all user-interfaces will immediately support the new operations.

4 ACCESS THOUGH A GUI

We have developed an X11/Motif GUI for VIPS which exposes demand-driven processing to the user. Figure 3 shows the interface being used on a 200MB image of a painting by Uccello. The L^* channel is being extracted and scaled to 0–255. It is being passed through a Laplacian, then thresholded, then single pixels are being removed with a morphology. As the controls in the windows are manipulated, the 50MB image on the right changes interactively.

5 COMPARISON WITH OTHER SYSTEMS

There are two earlier systems which are similar in some ways to VIPS. VPL[3], developed by Canon, is a user-interface which supports demand-driven computation. However, VPL is demand-driven at the level of the macro language, not at the level of the image processing operations — although the macro language is lazy, the operations themselves are strict.


ip
File Edit Shrink stdenv Helene Boolean Build Extract Convert Complex David
Arithmetic Statistics Rotate Grab Mosaic Histogram Graph Colour Relational
Balance Match Freqfilter Freqmask Fourier Help
Convolution A7
File View Options Help
Morphology A11
File Options Help
A13: A12 eor A9
File Edit View Options Help

A2	uccello.hr.v, FMTUCHAR, 8765x6329, 4 bands
A3	LabQ_to_Lab A2
A4	A3\1
A5	A4*2.55
A6	Convert_to_unsigned_char A5
A8	A7 A6
A9	A8 > A10
A10	148.49
A12	A11 A9
A13	A12 eor A9

A

No objects selected

Computed, FMTUCHAR, 8765x6329, 1 bands

282MB available

1	1	1
1	-8	1
1	1	1

Scale	Offset	Width	Height
1	128	3	3

	0	
0	1	0
	0	

Width 3 Height 3

1 band FMTUCHAR, NOCODING, 8765x6329 pixels, size 52.90 MB, 21.9x21.9 p/mm
(255 ,233)0 Mag x1

Figure 3: VIPS in action

Threads	User time	System time	Real time
1	125	4	148
2	126	6	96

Table 1: Timings for `calibrate`

Threads	User time	System time	Real time
1	13	5	45
2	14	5	44

Table 2: Timings for `bandjoin`

XIL[4], Sun's foundation imaging system, has a form of operation composition. Like VIPS, evaluation is delayed for as long as possible. When XIL finally does evaluate a list of operations, it first searches the list for combinations of simple operations which can be replaced by compound operations. XIL only has a limited number of these compound operations, and so most operation lists will not compose efficiently.

6 BENCHMARKS

This section presents the results of some simple benchmarks to show the efficiency of the system. These tests were run on a SPARCstation 10 with two 50MHz SuperSPARC processors. Output was always to `/dev/null`.

Table 1 shows the run times for `calibrate`, part of the VASARI calibration system. This program reads six 14MB images, corrects each for dark current and shading, normalises each one, performs aperture-correction with a four by four mask, transforms to XYZ using a six by three matrix, transforms to CIELAB, and writes the result. Each component of this program is written as a separate operation, joined together with VIPS partial image descriptors.

As can be seen, switching from one thread to two threads gives a speed-up of approximately 1.5. The acceleration is not by a factor of two, since there is a considerable amount of unavoidable delay reading the 84MB of input image from disc. To try to show how much of the run time was spent waiting for the disc, we timed another program, `bandjoin`, running on the same input files — see table 2.

`bandjoin` is almost entirely limited by disc IO, so there is no real speed-up moving to two threads. The times for `bandjoin` show that there are about 45 seconds of disc IO which have to happen. Looking back at `calibrate`, the one thread timing is 148 seconds — subtracting the 45 seconds spent waiting for disc gives 103 seconds of computation. Similarly for the two thread case, we have $96 - 45 = 51$ seconds of computation. The non-IO component has therefore been speeded up by a factor of two.

7 CONCLUSIONS

VIPS implements a fully demand-driven image IO system. This has the following important benefits:

- Simple image processing operations can be efficiently, flexibly and safely joined together to make more complicated operations. This composition works even if the images being processed are very large, since intermediate images are only ever partly there. It also works in the presence of simple coordinate transformations (such as convolution), and even radical transformations (such as 90° rotate).
- On machines with more than one CPU, image processing operations are automatically parallelised. Again, this parallelism works even in the presence of coordinate transformations. VIPS

implements simple mechanisms operations can use to produce accurate overflow counts, even when parallelised.

- Composition frees image processing operations from the need to know the details of the image file format. For example, in a few lines of C, `im_compress()` and `im_uncompress()` could be wrapped around the `invert()` operation of section 3, making a version of `invert()` which could directly process compressed images. VIPS uses exactly this technique to handle TIFF images.

VIPS has been used for building several large applications — including several image acquisition systems, two general user-interfaces, and a package for automatically mosaicing infrared reflectograms. VIPS is currently in use in museums and universities in America, Canada, the UK, Spain, France, Germany, Italy and Greece.

8 ACKNOWLEDGEMENTS

The MARC and VASARI projects were funded by the European Commission's ESPRIT programme. Other institutions involved in VASARI were: The Doerner Institut, Munich (D), Brameur Ltd. (UK), Telecom Paris (F), Thomson-CSF LER, Rennes (F), TÜV-Bayern, Munich (D). Other institutions involved in the MARC project were: Thomson Broadband Systems (F), The Doerner Institut, Munich (D), Crosfield Ltd. (UK), Hirmer Verlag (D) and Schwitter (Switzerland).

References

[1] K. Martinez, J. Cupitt and D. Saunders, 'High-resolution colorimetric imaging of paintings', *SPIE Vol. 1901 Cameras, Scanners, and Image Acquisition Systems*, pp. 25 – 36, San Jose, 1993.

[2] H. Derrien, 'MARC: A New Methodology for Art Reproduction in Colour', *Information Services and Use*, Vol. 13, No. 4, pp. 357 – 369, 1993.

[3] A. Davidson, P. Otto, D. Lau-Kee, 'Visual Programming', *Interacting with Virtual Environments*, ed. L. MacDonald and J. Vince, pp. 27 – 42, Wiley, London, 1994.

[4] Sun Microsystems, *XIL Programmer's Guide*, 1994.