https://doi.org/10.2352/EI.2025.37.12.HPCI-178

The libvips image processing library

John Cupitt; libvips.org; London, UK
Kirk Martinez University of Southampton; Southampton, UK
Lovell Fuller; Styling Ltd; London, UK
Kleis Wolthuizen; wsrv.nl; Sneek, NL

Abstract

libvips is a LGPL licensed (open source and free for commercial use), portable, horizontally-threaded, demand-driven, 2D image processing library with its origins in imaging research projects. Compared to similar libraries, libvips runs quickly and uses little memory. It supports numeric formats from 8-bit integer to 128-bit complex, any number of color separation bands, most popular image formats, and many specialized scientific image formats. It has become popular in applications such as virtual microscopy and art imaging, and very popular as an image processing library for the web.

This paper outlines the history of the library, explains how libvips achieves its good performance, presents benchmarks, and gives an overview of the implementation and of the wider libvips ecosystem.

Introduction

In 1989, as part of the VASARI project [1,2], we set out to make high-resolution, multispectral images of paintings. At 10 pixels per millimeter and with seven color separation bands, a scan of a 1m^2 painting would be 700 MiB; a huge challenge in an era when an expensive workstation might only have 32 MiB of RAM. There was no suitable existing open source package so we made our own: VIPS (VASARI Image Processing System) [3,4,5]. We designed it to cope with high resolution images (up to 2G x 2G) and with pixels of numeric types varying from 8 bit integers to 128 bit complex floating point numbers. Images could have any number of bands and could represent colors in one of many supported CIE color spaces.

Many image processing systems operate on whole images at a time, as represented conceptually in figure 1.

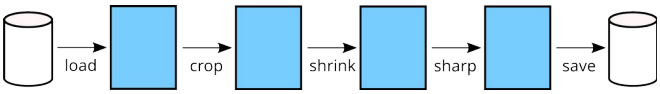


Figure 1. *Evaluation in a typical image processing library*

Reading from the left, first the data is read from storage and decompressed to a large memory area, then processed, perhaps in several stages, then finally recompressed and written back to storage again. The sequence of operations is strictly serial, that is, each operation must complete before the next can start. Intermediate images are memory arrays, so each operation needs at least two large areas of memory to function. And because the images will generally be larger than the processor caches, every pixel in every operation must be expensively fetched from main memory and written back again.

A design like this would not have worked for us. Our need to process images much larger than available memory forced us to adopt a stream processing model. We process images in small pieces, loading them one at a time from the source image, passing them through a chain of operations, and finally writing each piece to the output. Moreover, we needed to be able to perform coordinate transformations as part of the pipeline, so this processing had to be demand-driven. As the output file was written, each small area of the output image had to pull the required pixels through the system. Performing a series of operations on a small tile has the extra benefit that each operation will often find its input data already in the CPU cache, avoiding slow round trips to main memory.

Figure 2 shows libvips evaluation conceptually: all operations overlap, and whole images are never kept in memory. It shows a hard drive as the storage medium, but libvips has a generic IO layer, so the source and destination can be anything: a display, an area of memory, a pipe, or even an http connection to a cloud service.

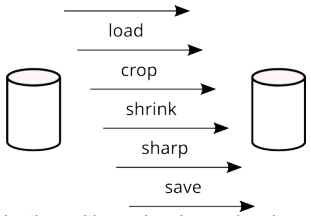


Figure 2. *libvips evaluation, with overlapping and no large intermediates*

When the first multi-CPU workstations arrived in the early 90s, we took advantage of this extra power by giving each CPU a lightweight copy of the whole pipeline, allowing each thread to generate an independent section of the output image. CPUs synchronized during the initial read from storage, and coordinated again during write, but otherwise ran largely independently, with only one lock operation for each section read and written, no matter how long the pipeline of processing operations.

By contrast, systems which operated on whole images at a time were forced to add threading in an ad hoc way in the implementation of each operation. As an operation executes, the set of available threads work cooperatively down the image, computing pixels. This arrangement forces threads to constantly synchronize, making them markedly less efficient than libvips.

By 2010, although the overall design seemed sound, the libvips implementation had become dated and hard to work with. We did a complete rewrite in the modern GObject C framework [6], adding introspection and a new, cleaner API. SIMD support was improved by adopting Google's Highway library [7]. We implemented support for a wide variety of useful image formats: JPG, PNG, TIFF, PPM/PGM/PBM/PFM, Radiance (HDR), JXL, FITS, NIfTI, SVG, GIF, Matlab, Analyze, OpenSlide, OpenEXR, WEBP, HEIC, AVIF, CSV, RAW and PDF. We also support load and save via the ImageMagick [8] (or optionally GraphicsMagick

[9]) library, so libvips can read and write any image format that these libraries can handle, such as DICOM.

This new foundation has let us greatly expand the range of language bindings we can support. Because the bindings use run-time introspection to present the operations they discover in the libvips binary, they automatically update with new versions of libvips, making them small and easy to maintain. The Python binding, for example, is written in pure Python and the core is less than 200 lines of code. The documentation component of these bindings is also generated largely automatically by introspection. This emphasis on easy and low-maintenance binding has helped libvips to quickly become one of the more popular image processing libraries.

libvips has a large test suite and is tested continuously with a range of static and dynamic sanitizers. OSS Fuzz [10], Google's fuzzing program which carries out continuous fuzzing of open source projects deemed to be vital infrastructure, has been testing libvips since 2019. It has found only one serious CVE in that time.

Benchmarks

We made a small benchmark to illustrate the benefits of a streaming design. It reads a 10,000 x 10,000 pixel uncompressed tiled RGB TIFF, crops 100 pixels off each edge, shrinks by 10% with linear interpolation, sharpens with a 3x3 convolution and then writes the results back to another TIFF. This is a simple test, but it is easy to implement in almost all image processing systems, and exercises IO, resampling, filtering and coordinate transformation – the basic operations any system should be able to do well.

We implemented this test with a range of libraries and timed them running on an AMD Ryzen Threadripper PRO 3955WX 16-Cores, with Ubuntu 24.04 in performance mode. The benchmark runs each test five times and takes the best result. The results are shown in table 1 and are available in full on the git repository [11].

Table 1. Speed and memory use for a range of image processing systems

Software	Time (secs)	Peak mem (RSS MiB)	Times slower
libvips 8.16, C/C++	0.46	91	1.0
libvips 8.16 python	0.49	101	1.1
tiffcp	0.59	581	1.3
libvips 8.16, one thr.	0.72	52	1.6
libvips 8.16, JPG	0.80	189	1.7
Pillow-SIMD 9.0. one thr.	1.49	985	3.2
GraphicsMagick 1.4	1.97	1989	4.3
ImageJ 1.54	2.70	542	5.9
OpenCV 4.6	3.03	791	6.6
ImageMagick 6.9.12	3.57	2002	7.8
libgd 2.3.3, JPG	5.41	4268	6.8
GEGL 0.4.48, JPG	5.74	751	6.1
NetPBM 11.05	6.65	684	14.5

libvips is at least twice as fast as any other system tested and typically needs five to ten times less memory.

Figure 3 shows the same data as memory use over time. libvips is the small bump in the bottom left corner. The github repository for this benchmark has the code to generate this graph.

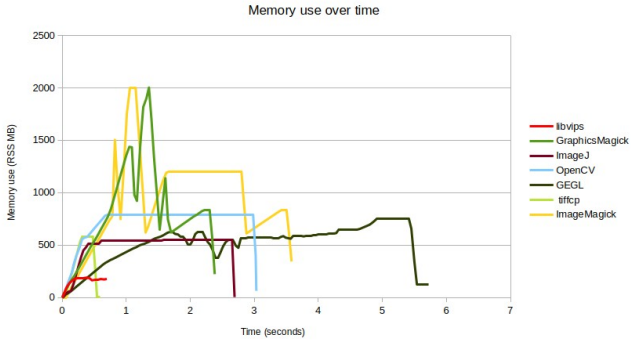


Figure 3. *Memory use over time for a range of image processing libraries*

The wider libvips ecosystem

A range of downstream projects have appeared around libvips, from language bindings to frameworks, desktop applications and web services.

pyvips, ruby-vips, php-vips, lua-vips

These bindings are all fully dynamic, and have no native component, other than libvips and an FFI module. They define an image class, then intercept the missing method exception to call into libvips using a small marshaling class to turn language values into and from the GValue types needed by libvips. They usually define a few convenience methods too, for example a set of operator overloads.

The Python interface is currently downloaded 50,000 times per month (2024) and is popular for art and for medical imaging. It interfaces easily with other popular Python packages, such as PIL and Numpy. The Ruby binding is now the default image handler for Rails, the popular web application framework, and is downloaded almost a million times a month (2024).

cplusplus-vips, NetVips, go-vips, crystal-vips

Bindings for more static languages tend to come in two halves. They have a class which can dynamically call any libvips operation by name, and then a set of very thin wrappers over that, generated at compile-time from introspection, which provide static type checking. Maintenance is slightly more work than the fully dynamic bindings, but usually just involves running a small script before each release. NetVips, the C# binding for libvips, has become reasonably popular, with over 10,000 downloads per month.

Wasm

Wasm is a relatively new way of running native code from JavaScript. C/C++ sources are compiled to an intermediate binary form (called Wasm, for WebAssembly) with a compiler like Emscripten, then at runtime the JavaScript implementation transforms the Wasm code to a fully native binary.

Performance is reasonable, a Wasm build of libvips is faster than a native build of ImageMagick for example, but between two and four times slower than a native build of libvips, mostly due to the use of platform-specific intrinsics in many existing libraries. The complete libvips binary, including a useful selection of image format libraries, compiles to under 5 MiB of Wasm.

Sharp

Sharp is a node package using libvips to implement high-performance image resizing for all Node.js compatible JavaScript runtimes, including Deno and Bun. There was an early focus on fluent API design, partly matching the underlying libvips C++ API, with some additional simplification for common tasks. A couple of features originally added to Sharp have migrated upstream and are now integrated within libvips itself, namely image composition and smartcrop.

Sharp started life as a Responsive Web Design experiment. Previously, when images were uploaded to web sites, they would be resized to a small number of fixed dimensions and then served to clients from those static resources. Thanks to libvips, Sharp is fast enough that it can generate resized images on demand. Websites not only see a significant saving on storage costs, they can also serve images exactly tailored to the client device, something that is becoming increasingly important as the web becomes slowly more heterogeneous.

The rise in popularity of Node.js as a server-side JavaScript runtime led to a rise in the popularity of Sharp. Gatsby, the progenitor of the "Jamstack" web architecture, selected Sharp for all of its image processing needs. In the subsequent years, almost all other JavaScript-based web frameworks and Content Management Systems have followed in adopting Sharp. Both Amazon Web Services (AWS) and Google Cloud recommend Sharp for image resizing as part of Lambda and Cloud Functions, their respective JavaScript function-as-a-service offerings.

In 2023, Sharp was downloaded from the npm JavaScript registry over 150 million times. In 2024, it is predicted that Sharp will be downloaded at least 250 million times. It is currently the most popular downstream consumer of libvips by some margin.

To make installation easier, prebuilt binaries are provided by Sharp for itself, libvips and around 25 other upstream dependencies, primarily those that allow it to support encoding and decoding of image formats popular on the web. This lets Sharp install painlessly on a wide range of platforms, from a Raspberry Pi Zero to an IBM s390x mainframe. Sharp can also be used with a Wasm build of libvips, enabling complete portability. Its uptake has been shown to reduce the CPU load on provider's cloud machines.

nip2 and desktop applications

Several desktop applications use libvips as the image processing engine. The official libvips GUI is nip2, a spreadsheet-like image processing application.

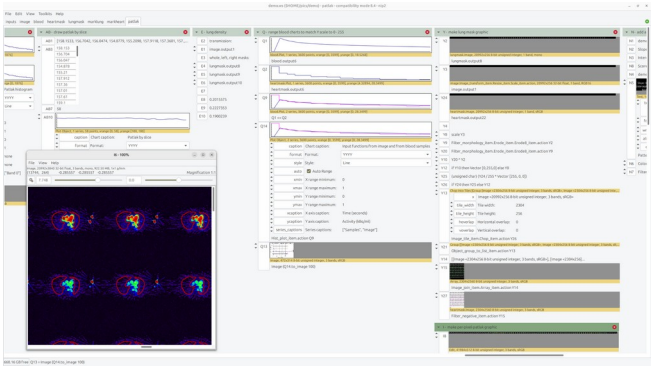


Figure 5. *Dynamic modeling of FDG uptake in a pulmonary PET-CT scan using nip2*

The user can develop an image processing application by loading images into cells and entering formulas or selecting a formula from menus. Thanks to libvips, memory use is low and recomputation is very fast and completely on-demand. Formulas can be adjusted interactively and the pixels change in all the derived images. Once a prototype is working, it can be executed in batch mode over a large set of inputs and without a graphical interface.

The workspace shown in figure 5 computes per-voxel rate constants for FDG uptake in a pulmonary PET-CT scan. It contains around 9,000 cells, and over 20,000 images, totaling over 150 GiB. It takes 30s to fully recalculate on a desktop PC (though small changes are much faster) and runs in 2.5 GiB of memory.

Art imaging

After the VASARI project, a number of art imaging projects have used libvips as their processing engine.

Operation Night Watch [12], a project at the Rijkmuseum in The Netherlands, has used libvips to assemble and process their 717 gigapixel image of Rembrandt's *The Night Watch*, at the time the largest digital image of a work of art ever assembled.

This colossal image was shortly superseded by a 1.6 terapixel scan of *The Panorama of the Battle of Murten* [13], a panoramic scroll by Louis Braun, held in Geneva. This currently holds the record for the largest digital image of a work of art at 3,828,940 by 440,000 pixels.

Related software

A great many image processing frameworks have been developed over the decades and this paper does not attempt a full survey or comparison. Systems like OpenCV [14] and ImageMagick [8] use the whole image at once design. They represent images as huge memory arrays and have sets of carefully optimized functions for transforming these arrays in various ways. Threading is ad hoc and implemented afresh each time in each processing function. There is little support for tiling or for large images, and memory use is high. ImageMagick does have some limited support for image streaming.

Another family of systems rely heavily on C++ for genericity. Eigen [15], CImg [16] and pylena [17] are all largely compile-time systems which attempt to generate optimized code for an image processing pipeline using C++ templates. They do not work well when called from more dynamic languages, usually need explicit coding for lazy computation or tiling, are tricky to use on very

large images, and have threading support coded in an ad hoc manner for each operation.

There are some historical systems which are fairly close to libvips. SGI had the ImageVision library [18], and Pixar had a 2D animation image composition system [19]. NASA developed SIPF [20] (Scalable Image Processing Framework) for processing very large images from planetary missions. It was demand-driven, but tile-based and designed for computation to be distributed over a large cluster of machines, making interactive use cumbersome. To our knowledge, none are open source, maintained or available.

GEGL [21] is perhaps the project that is closest to libvips, but it threads vertically and is designed for interactive image editing with very heavy emphasis on caching. This makes it a poor fit for batch-style processing, as can be seen in the Benchmarks section above.

Implementation

libvips has a small, simple and portable implementation with less than 20,000 lines of C for the core of the system, plus another 160,000 in a mixture of C and C++ for the image processing operations.

At the lowest level of the system, libvips tracks pixel buffers: small, rectangular parts of an image. It uses small (only two or three buffers are kept per image), thread-private (that is, unsynchronised) caches on images based on hashes of pixel coordinates to discover sharing and to reuse pixel buffers between different parts of the image processing pipeline.

Regions sit at the next level up. Again, these are rectangular areas of an image, but as well as simple pixel buffers, they can also be backed by pixels on other regions, or even on other images. Consider an operation which places a small image on top of a much larger image at some position. Regions which contain an overlap area will need to be represented by pixel buffers, but regions which are entirely within either the large or small image can simply be references to those images. Regions let operations implement things like crop, copy and translate with references rather than real memory. This is important for performance in systems like libvips where all operations are non-destructive and repeated copying could become a significant bottleneck.

Stepping up the hierarchy again, a partial image is one where, instead of storing a value for each pixel, libvips instead stores a function which can compute any region on demand. When an operation requests a region on a partial image, libvips will select and size a pixel buffer to hold the requested pixels and use the stored function to calculate just those values (or reuse them, if a valid pixel buffer already exists).

The stored function comes in three parts: a start function, a generate function and a stop function. The start function creates a state, the generate function uses the state plus a requested area to calculate pixel values, and the stop function frees the state again. Breaking the stored function into three parts is good for thread scaling: resource allocation and synchronization mostly happens in start and stop functions, so generate functions can run without having to synchronize or allocate resources. libvips makes a set of guarantees about parallelism that make this simple to program. Start and stop functions are mutually exclusive and a single state is never used by more than one generate operation. A start / generate / generate / stop sequence works like a thread, as illustrated in figure 4.

Above images sit operations. These define generate functions on images which accept requests for regions of pixels, in turn request regions from their inputs, and then compute results. The region create / request / free calls used to calculate pixels on an image are an exact parallel to the start / generate / stop calls that partial images use to create pixels. In fact, they are the same, and this is the composition mechanism that libvips uses to join operations together. A region on a partial image holds the state created by that image for the generate function that will fill the region with pixels.

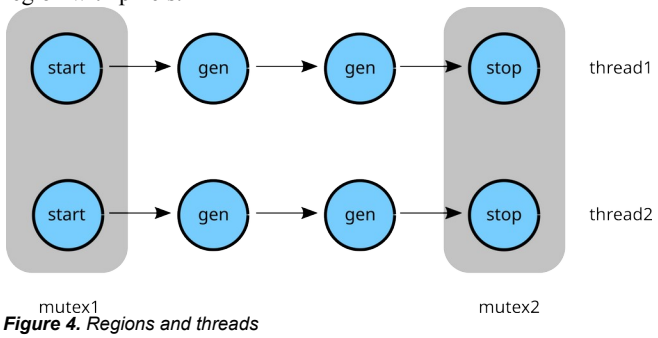


Figure 4. *Regions and threads*

libvips joins image processing operations together by linking the output of one operation (the start / generate / stop sequence) to the input of the next (the region it uses to get pixels for processing). This link is two function calls plus a hash table lookup, and is very fast. Demand flows from sink to source as a series of function calls between the generate functions of the operations, and pixels flow back from source to sink in a chain of pixel buffers computed as the stack unwinds.

When reading from a large libvips image (or any other format with the same structure on disc, such as binary PPM), libvips keeps a set of small rolling windows into the file, some small number of scanlines in height. As pixels are demanded by different threads, libvips will scroll these windows up and down the file.

Some operations, such as a 90 degree rotation, need random access to source pixels. If the image format does not support random access (PNG, for example), libvips will automatically unpack the image to a temporary area in memory or storage. If simple top-to-bottom sequential access is enough, then libvips will attach a queue and a small scanline cache to the image format read library. As threads request pixels, libvips will stall and reorder them to ensure access remains in order.

In a demand-driven system, something has to do the demanding. libvips has a variety of data sinks that can be used to pull pixels through a pipeline. There are sinks that will build a complete image in memory, sinks to draw to a display, sinks to loop over an image (useful for statistical operations, for example) and sinks to stream an image to storage.

The storage sink keeps two buffers, each as wide as the image. It starts threads as rapidly as it can, up to the concurrency limit, filling each buffer with regions of calculated pixels, each thread calculating one region at once. Regions can be any shape and size: libvips has a hint system that operations use to tell sinks which geometry they prefer. libvips watches the set of worker threads and automatically sizes thread pools up and down with load to minimize stalling.

A separate background thread watches the buffers and, when they fill, writes that set of scanlines to storage using whatever image write library has been selected. It then resets the buffer and moves it down the image, ready for the next set of pixels to stream in.

These features in combination mean that, once a pipeline of image processing operations has been built, libvips can run almost lock-free. This is very important for SMP scaling: we did not want the synchronization overhead to scale with either the number of threads or the complexity of the pipeline of operations being performed. As a result, libvips scales almost linearly with increasing numbers of threads, provided the pipeline has sufficient parallelism.

Some image processing operations cannot be implemented efficiently in this streaming style, for example Hough or Fourier transforms. Operations like this signal that they need access to all pixels at once and libvips will arrange for the image to be rendered into memory or to a temporary image in storage. Operations such as flood-fill are inefficient without a destructive image write. These functions signal this requirement to libvips and those sections of the pipeline are enclosed in a non-destructive wrapper.

By default, libvips does very little pixel caching. This is usually correct for batch processing on large data sets. However, interactive applications often reuse the same area of an image, perhaps as the user moves a slider in an interface, and caching becomes essential for good performance. A number of operations implement different types of cache, from sequential line caches to threaded, sparse tile caches, and can be explicitly added to pipelines as required.

Because libvips operations are free of side-effects, results can be reused, a technique usually called memoization. Every time an operation is called, libvips searches an operation cache for a previous call with the same arguments, and if it finds a match, returns the previous result. By default, libvips caches the last 1,000 operation calls or 100 MiB of memory, whichever is smaller.

libvips has around 300 image processing operations written in this streaming style. The library supports full run-time introspection using GObject, so the standard GObject calls can be used to walk the class hierarchy and discover operations. libvips adds a small amount of extra introspection metadata to handle things like optional and deprecated arguments.

Limitations

As well as strengths, the libvips demand-driven and tile-less design also has limitations.

Many image processing operations, for example a Fourier transform, require the complete image to function. libvips supports operations like this by rendering the input pipeline to a large memory image, evaluating the operation and then executing the pipeline after, but the flow has obviously been interrupted. There will be a large increase in memory use, and, potentially, a large drop in parallelism.

Similar problems arise with operations that need significant local context, perhaps a Gaussian blur with a large radius. Because libvips has no tiles, it has no tile caching, and this will cause over-computation on region boundaries. The over-computation cost is often not worth worrying about. If it does become an issue, users can always add an explicit tile cache operation beforehand.

There is also a more general limitation: for the most part we have only implemented operations we've needed in our work, so users from other domains may not find the tools they need. In this case, they would need to implement new library features. libvips is at least open source and well-documented.

Conclusions

libvips achieves high performance and efficiency thanks to careful design, driven by the early days of high resolution imaging. The LGPL license (open source with free commercial use) has encouraged participation and continuous development. It has been especially helpful in finding volunteers to package libvips for most popular operating systems, making libvips widely available.

Since its development in the 1990s, libvips has been used as the image processing package in five European projects, completely rewritten and modernized, and is now widely used in research and on the web. It is used in large applications where pipelines contain well over 10,000 nodes. It is the most popular image processing library for node.js with almost 200,000 downloads per day, it is the default image processing library in Rails, and companies from Amazon and Shopify to Wikipedia use it for image handling.

In the beginning, libvips was designed for single, multi-gigabyte images. Today, it is more likely to be used for processing millions of mobile phone photographs or images too large for other packages. Its original virtues of low memory footprint and efficient processing have remained relevant. The continuous growth in core-count in CPUs automatically leads to speed increases due to our early developments in the 1990s.

Acknowledgments

We would like to thank the following contributors and funders of libvips: The European Commission for its financial support in the projects VASARI, MARC, ACOHIR, Artiste and SCULPTEUR. Various companies and individuals through open collective. Nikos Dessipris for early developments and Ruven Pillay for developments including autoconfiguration. Jean Philipe Laurant wrote many of the color functions. Thanks also to David Saunders, the first real user of the system, for his feedback over the years.

References

[1] K. Martinez, J. Cupitt, D. Saunders, "High resolution colorimetric imaging of paintings", Proc. SPIE, Vol. 1901, pp. 25-36, 1993.

[2] K. Martinez, J. Cupitt, D. Saunders, and R. Pillay, "Ten years of art imaging research". Proc. IEEE 90, pp. 28-41, 2002.

[3] J. Cupitt and K. Martinez, "VIPS: an image processing system for large images", Proc. SPIE conference on Imaging Science and Technology, San Jose, Vol. 2663, pp. 19-28, 1996.

[4] K. Martinez and J. Cupitt, "libvips - a highly tuned image processing software architecture", Proc. IEEE International Conference on Image Processing 2, pp. 574-577, Genova, 2005.

[5] The libvips website, libvips.org, accessed July 2024.

[6] The GObject website, docs.gtk.org/gobject, accessed July 2024.

[7] Highway website, github.com/google/highway, accessed July 2024.

[8] ImageMagick website, imagemagick.org, accessed July 2024.

[9] GraphicsMagick website, graphicsmagick.org, accessed July 2024.

[10] OSS Fuzz website, github.com/google/oss-fuzz, accessed July 2024.

[11] libvips benchmark, github.com/libvips/vips-bench, accessed July 2024.

[12] Operation Nightwatch, rijksmuseum.nl/en/whats-on/exhibitions/operation-night-watch, accessed July 2024.

[13] Murten Panorama Digital Twin Scanning Project, paulbourke.net/panorama/MurtenStory, accessed July 2024.

[14] The OpenCV website, opencv.org, accessed July 2024.

[15] The Eigen website, eigen.tuxfamily.org, accessed July 2024.

[16] D. Tschumperle, C. Tilmant, V. Barr, “Digital Image Processing with C++, Implementing Reference Algorithms with the CImg Library”, CRC Press, 2023.

[17] M. Roynard, E. Carlinet, T. Géraud, “An Image Processing Library in Modern C++: Getting Simplicity and Efficiency with Generic Programming”, *Reproducible Research in Pattern Recognition: Second International Workshop, RRPR,* 2018.

[18] A description of the SGI ImageVision library, wiki.preterhuman.net/ImageVision_Library, accessed July 2024.

[19] M. Shantzis, “A model for efficient and flexible image computing”, SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques, 1994.

[20] M.W. Powell, R.A. Rossi, K. Shams, A scalable image processing framework for gigapixel mars and other celestial body images", IEEE Aerospace Conference, 2010.

[21] The GEGL website, gegl.org, accessed July 2024.

Author Biography

John Cupitt received his PhD in Theoretical Computer Science from the University of Kent (1989), then spent 15 years at The National Gallery in London working on art imaging, and 15 years at Imperial College, London working on medical imaging. He is currently a freelance developer.

Kirk Martinez has a PhD in parallel image processing architectures from the University of Essex and is a professor in Electronics and Computer Science at the University of Southampton. He has designed many imaging systems for museums through ten EU projects and is currently focusing on environmental sensor networks.

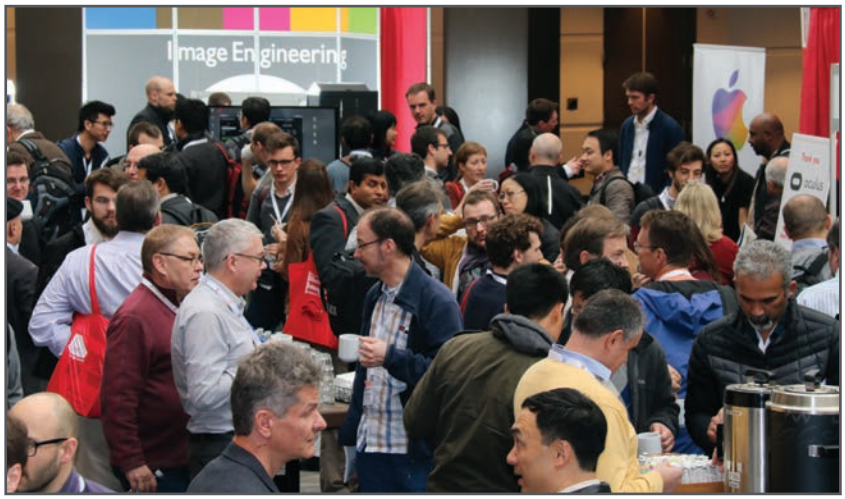
Lovell Fuller has a BSc in Computer Science from the University of East Anglia. He has 25 years commercial experience in developing software for the web as both a team member and leader. He is responsible for creating and maintaining popular, open source image processing software used widely within the JavaScript ecosystem

Kleis Auke Wolthuizen has a Software Engineering degree from NHL Stenden, University of Applied Sciences in Leeuwarden. He maintains the Windows binaries and various bindings for libvips. He manages wsrv.nl, a free and open-source image caching and resizing service, which processes 6 million images per hour on-the-fly using libvips.

JOIN US AT THE NEXT EI!

electronic IMAGING

Imaging across applications . . . Where industry and academia meet!



• SHORT COURSES • EXHIBITS • DEMONSTRATION SESSION • PLENARY TALKS •
• INTERACTIVE PAPER SESSION • SPECIAL EVENTS • TECHNICAL SESSIONS •

www.electronicimaging.org

ADVANCING IMAGING SCIENCE 75 1947 IS&T 2022 imaging.org