https://doi.org/10.2352/ISSN.2169-2629.2018.26.383
© 2018, Society for Imaging Science and Technology

LittleCMS-MT: A thread-safe open source color management library

Michael J. Vrhel, Robin Watts and Ray Johnston, Artifex Software, 1305 Grant Avenue, Suite 200, Novato, CA 94945

Abstract

We introduce a thread-safe color management library, LittleCMS-MT. It is a fork of the popular LittleCMS color management library. The issues with LittleCMS are described and the approach used to create LittleCMS-MT is discussed. The architectural design of using a thread-safe color management library with open source projects used for rendering page description languages is covered. Performance results using Ghostscript with LittleCMS-MT vs. LittleCMS and standard test files are provided.

Introduction

LittleCMS[1] is a popular open source color management engine. It was introduced in 1998 by Marti Maria and currently is provided under an MIT license. The current version number is 2.9, which was released in November of 2017.

LittleCMS has been used for years in Ghostscript[2], which is a dual licensed (GNU AGPL and commerical) open source project used for the rendering and conversion of PostScript®[3], PDF[4], XPS[5] and PCL[6] documents. Ghostscript was introduced in 1986 by L. Peter Deutsch. It is currently at version 9.23, which was released in March of 2018. Today, Ghostscript is embedded in millions of devices/applications including printers (desktop and production), RIPs, document management software systems, document viewers and cloud applications to name a few.

One of the limitations of LittleCMS is that it is not thread-safe. While it is true that LittleCMS can be used in multi-threaded applications, these applications must be designed to use LittleCMS in a way that achieves a less than ideal level of performance. For example, in LittleCMS, it is not possible to safely share ICC profiles or links (structures that define mappings between ICC profile device color spaces) between threads without placing a lock around them during their use. This severely limits the performance of applications such as Ghostscript, which are commonly used in a multi-threaded manner and have heavy color conversion needs.

To overcome this limitation, a thread-safe version of LittleCMS has been released under the same MIT license[7]. The new version is referred to as LittleCMS-MT. In addition to providing a thread-safe color management library, LittleCMS-MT provides accelerations that are currently unavailable in LittleCMS. The motivation for this work and its efficient use is described in the remainder of this paper.

Thread safety

One of the features of a library like LittleCMS is the ability for the library to call back to the calling application. These callbacks are typically used to provide error trapping, memory management or to provide processed data in an asynchronous manner. A requirement in designing a thread-safe library that uses callbacks is that the callback routines should be capable of accessing the variables and data structures required to work. The standard manner in which this is usually achieved is to allow the caller to pass in an opaque pointer as one of the arguments to every procedure in the library's application programming interface (API), which will then be passed back unchanged in every callback.

With the thread's context propagated through the library functions, all callbacks can then provide a pointer to the state or context associated with the thread. This ensures that the calling thread does not receive pointers to structures for which it should not have access and (assuming there are not other issues such as global variables etc.) avoids contention or race conditions between threads.

The existing LittleCMS library achieves some level of this design as it provides the concept of a contextID to be passed with some of its procedures. Unfortunately, the design falls short as the contextID is not passed to all of the procedures. In addition, LittleCMS stores the contextID into some of its structures effectively cementing those structures to be used only by the calling thread that created them. This situation is shown in Figure 1, where Thread1 creates a resource *rsrc1* (for example an ICC profile structure or a transformation structure) using its context *ctx1*, which LittleCMS stores inside of *rsrc1* . The object *rsrc1* is then stored in a cache so that it can be used by other threads. To avoid the overhead of creating its own copy of *rsrc1*, Thread2 makes use of the *rsrc1* object that is present in the cache and passes it along with the thread's context *ctx2* to the LittleCMS library. Due to its less than optimal design, the library fails to propagate *ctx2* through its calls and at some point makes use of the context *ctx1* stored in the resource structure, passing *ctx1* to Thread2 in a call back. Since Thread1 and Thread2 are now both possibly making changes to *ctx1* (or using it to access certain memory addresses), thread safety is no longer ensured.

In LittleCMS-MT, the above problem is avoided by having the incoming contextID propagated to all methods within the library, including static ones. In addition, storage of the caller's contextID in structures within the library is not allowed. In this way, a thread will receive only its own contextID on any callback. This situation is shown in Figure 2, where after an API call to the library with api(*ctx2*) from Thread 2, any callbacks that occur must, by design, pass back *ctx2* and can never return *ctx1*.

Ghostscript

Ghostscript's architecture is shown coarsely in Figure 3. At the top of this figure, we have the interpreters for the various page description languages (PDLs) that Ghostscript supports. At the bottom of the figure we have the output device and list a few of the many output formats that Ghostscript supports. These output formats include other PDL formats as well as image formats

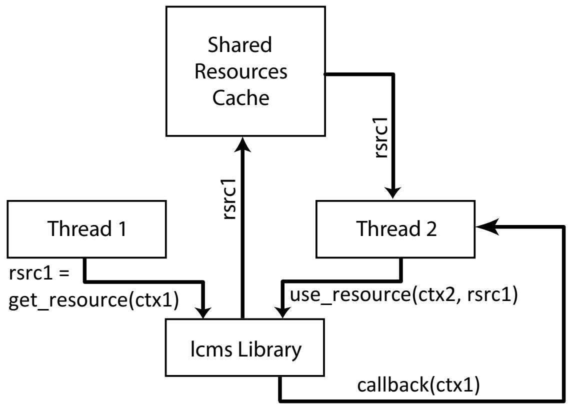

Figure 1. *Diagram of unsafe thread interaction when using LittleCMS*

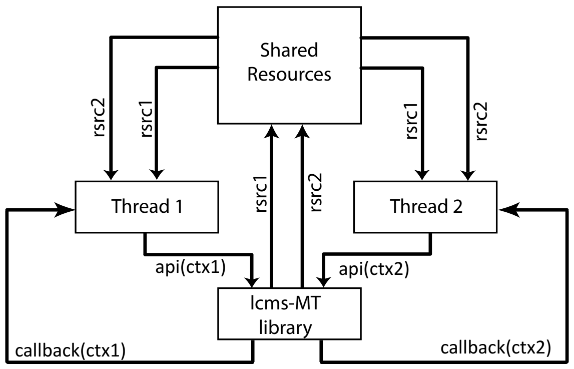

Figure 2. *Diagram of safe thread interaction when using LittleCMS-MT*

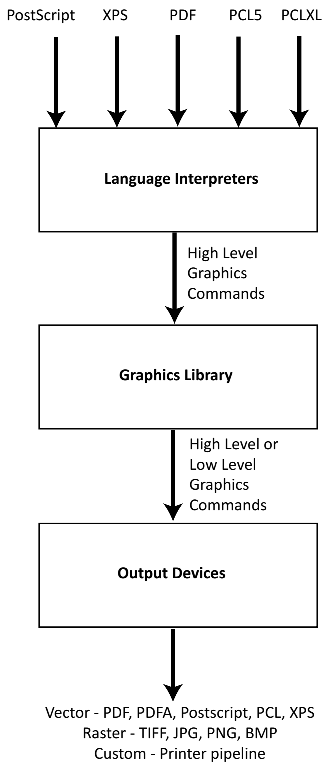
PostScript
XPS
PDF
PCL5
PCLXL
Language Interpreters
High Level Graphics Commands
Graphics Library
High Level or Low Level Graphics Commands
Output Devices
Vector - PDF, PDFA, Postscript, PCL, XPS
Raster - TIFF, JPG, PNG, BMP
Custom - Printer pipeline

Figure 3. *High level view of Ghostscript's architecture*

and customized printer driver formats. Between these two components is the graphics library. When going from one PDL format to another PDL format (e.g. PostScript® to PDF) the output device may understand how to handle a high level drawing command and the graphics library may not need to do any work. Instead the output device simply repackages the command in the appropriate manner for the output PDL format. If instead, we are going out to an image format, the graphics library will break down the higher level commands to lower level commands (e.g. to a rectangle fill or a bitmap copy).

To provide proper rendering, Ghostscript must be able to handle a variety of managed color definitions. These include PostScript® color space arrays, ICC profiles (both V2 and V4), CIELAB, PDF CalGray and PDF CalRGB color spaces. In addition, PostScript® and PDF have generic Gray, RGB and CMYK color spaces. These are typically assigned with default ICC profiles that are run-time configurable by the user. To simplify things, Ghostscript converts all non-ICC managed color definitions to equivalent ICC profiles.

A PDL document can contain text, vector graphics and images each of which can be defined in different color spaces. Rendering to a target, Ghostscript allows the user to define a single target ICC profile or unique profiles for image, graphic and text objects. This flexibility of color space definitions for different object types can lead to documents that have significant color conversion requirements. For an embedded PDL rendering engine, rendering performance is always a concern as it has an obvious impact on the printer page-per-minute specification. Creation of links between ICC color spaces can be a significant computational

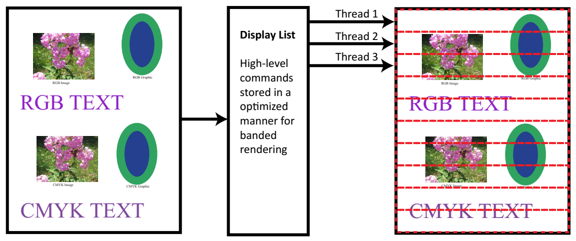

Figure 4. *Display list multi-threaded rendering*

cost, especially if larger table sizes are created. To minimize the number of links created, Ghostscript maintains a cache of previously created links, which is searched when a link is needed by a thread. If the needed link is not found in the cache, a new one is created and stored in the cache.

When rendering to high resolutions, Ghostscript will store the various page drawing commands into a display list. Part of the description for these drawing commands are the associated ICC color spaces, which are also stored in the display list. Once created, this display list can then be processed to render the graphic, text and images onto the output bitmap. Use of the display list enables the data to be structured in a way that optimizes for only rendering a small band at a time, thereby saving on memory needs for the system. When rendering in a single threaded manner, Ghostscript will first render the top band, then the next, all the way to the final band. When Ghostscript is operated in a multi-threaded manner, each thread can independently render a band. This parallelism can provide a significant speed-up when using a multi-core processor. Figure 4 shows an example where the output page is divided into multiple bands and three threads are each independently rendering different bands. When a thread completes a band, it will move to the next unprocessed band.

With multi-threaded rendering in Ghostscript, each thread maintains its own memory pool manager. The memory pools use a splay tree [9] to organize the memory chunks. Splay trees are essentially binary search trees that can reorganize themselves when an item is accessed. This reorganization can ensure more commonly used items are near the top of the tree and therefore more quickly accessed. If the threads had a shared memory pool, it would be neccessary to place a mutex lock around the splay tree operations to avoid concurrent problems. Use of a mutex in this case would result in a severe performance penalty. The thread's memory pool manager is the context information passed to LittleCMS-MT with every call to the library. When the library needs to do a memory allocation, it calls back to a memory allocation function in Ghostscript, which will then ensure that the allocation occurs with the proper memory pool manager.

In LittleCMS, the fact that the wrong memory pool manager can be passed back to a thread, means that concurrency issues can occur. To avoid this problem when using LittleCMS, structures such as links and profiles can not be shared between threads, and each thread must maintain its own cache of link transforms. This limitation can have a significant impact on performance. For example, consider a document with eight vertical strips on a page as shown in Figure 5, each defined with different ICC color spaces. If there are four threads processing bands on this page there would need to be 32 different links created. This can have an impact on

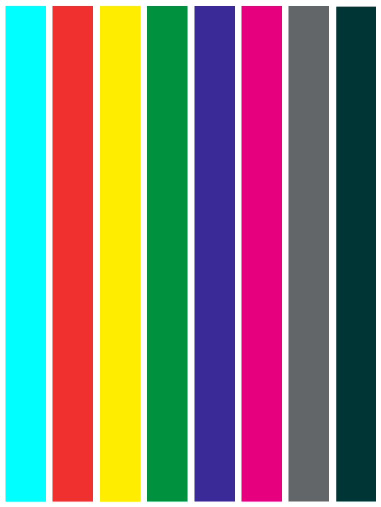

Figure 5. *PDF Document with eight CMYK rectangle fills each defined by a different ICC profile*

the performance boost that should occur when rendering the page with multiple threads on multiple cores. With LittleCMS-MT, it is possible to have the threads share a common cache. In this case, for the above example, only eight different links would need to be created. We will study the performance impact this has on standardized files in the results section below.

MuPDF

MuPDF[8] is an open source dual licensed (GNU AGPL and commercial) project designed for the rendering, manipulation, editing and converting of PDF, XPS and EPUB document formats. The code is designed to work well in mobile environments. It is currently at version 1.12, which was released in December 2017. To achieve managed color output, LittleCMS-MT was included in the project. In the case of MuPDF, the context passed to LittleCMS-MT and passed back during memory allocations includes a pointer to the thread's exception stack. Using LittleCMS in MuPDF with shared objects would lead to issues as one thread could alter the exception stack of another thread. The LittleCMS-MT library avoids this problem by ensuring the proper context is always provided in the callback to the thread.

LittleCMS-MT Optimizations

Ghostscript and MuPDF both require the ability of the color transforms to operate on a variety of different data formats. These different formats could all occur on a single page of a document. For example there could be color data that is 8-bit or 16-bit, planar or chunky image data, different endianness, and include/exclude an alpha component. Little CMS is designed to handle these various formats. The LittleCMS-MT project has introduced optimizations in this design, providing an increase in performance over LittleCMS.

Due to the cost of creating a link between ICC profiles, it is desirable to avoid creating an entire new link due to a simple input or output format change. Little CMS has a method to do this, but it is not thread-safe. Therefore LittleCMS-MT removes this method and provides a thread-safe solution.

Pack/unpack

The core LittleCMS buffer transformation routine takes a set of input values, and transforms them into a set of output values, with or without a cache, or gamut check. The exact format of the values in these buffers can be set by the caller to match the formats of the data most natural to the calling routines.

Little CMS provides interpolation routines that work in both floating point and 16-bit integer ranges. In order to accommodate the other input and output formats used, the LittleCMS buffer transformation code breaks the process down into various stages; 1) Unpack, 2) Cache Check, 3) Gamut Check, 4) Transform, and 5) Repack.

The unpack stage takes the data from the supplied buffer and converts it to the required internal format (normally 16-bit integers). The cache check compares this unpacked data against the previous input values, and if it finds that they match, can short circuit the rest of the process by reusing the previously computed values. The gamut check (a relatively rarely used option) checks to see whether the values are out of range for the profile, and if they are it copies in known "alarm" values, short circuiting the rest of the process. The transform stage performs the interpolated lookup of the input values to get the output values (both in the standard internal form). Finally, the pack stage copies this internal data back (converting if required) into the output buffer.

In Ghostscript and MuPDF, we always feed data in either 8 or 16-bit integer form. In the 16-bit case, the pack/unpack stages resolve to a copy operation and in the 8-bit case they resolve to a quickly performed operation.

In the original LittleCMS code, the packers and unpackers were held as function pointers, so every pixel processed would involve at least 3 function calls (one to unpack, one to transform, and one to pack again). Function call overhead for such trivial operations (or NOPs) is significant when taken in bulk, so a way to reduce this cost was sought.

Accordingly, we created a "templated" routine that can be used to generate optimized code for given color transformation, in the form of a header file that can be repeatedly included with different definitions. In this process, some defines are made, and then inclusion of them generates a new instance of a transformation routine optimized for the given formats.

We have deployed these optimized routines for all the common gray/RGB/CMYK in/out operations with both 8 and 16-bits. Operations on 16-bit data now operate "in-place" with no unpacking/packing required. Operations on 8-bit data have the pack/unpack operation performed directly, avoiding the function call overheads. If a user discovers cases that are not optimized already, adding new cases is a trivial matter of a few #defines, and another include.

ChangeBuffers vs. Cloning

Once a link has been created between given input and output formats, LittleCMS provides a mechanism for changing these formats without recalculating the entire link (for certain classes of input/output formats at least). For example, if a link is created to consume and return 16-bit data, but it is later discovered that the same link is required to handle 8-bit data, LittleCMS would allow you to change the buffer format without the expense of creating a new link and recalculating the internal tables.

The mechanism for this involved LittleCMS changing the pack/unpack pointers within a link. This causes problems for multi-threaded use, as one thread may be in the process of using a link when another changes the formats for which it is setup, causing undefined results.

Accordingly, in LittleCMS-MT, we remove this operation and replace it with a new operation to "clone" a link with different input/output formats. Both the clone and the original link share the internal tables in a reference counted, thread-safe fashion, thus avoiding the overhead of holding or calculating the internal tables multiple times.

In Ghostscript, a cached link object could contain 128 potential clones of one single link from the color management library. These clones share the same core object and have different input and output buffer settings to handle the variety of different input and output formats (e.g. planar/chunky data, 8/16 bit, big/little endian on the input and output settings as well as the inclusion/exclusion of an alpha value). The cloning occurs on an as-needed basis. The clones are maintained in a linked list and can be readily shared among threads.

Results

To measure the performance gain achieved by using LittleCMS-MT in Ghostscript, the PDF files shown in Figures 6-8 were rendered. The PDF file shown in Figure 6 consisted of six images in sRGB color space each of size 4608x3456 pixels and the PDF page size was 23.9 x 26.7 cm. We will refer to this file as the Images file. The PDF file shown in Figure 7 is the Altona 1.2 Visual test page. It contains a number of elements in various color spaces, the details of which are discussed in [10]. This PDF page size was 44.1 x 31.8 cm. Finally, the PDF file shown in Figure 8 is the Altona 2.0 test file [11] and is sized at 29.7 x 42.0 cm. This file provides transparency objects that can be difficult for some rendering solutions. Together, these files provide a wide variety of different color types and elements that can be encountered in a PDF file.

An Intel® 24-core Xeon® X5650 CPU 1.6-2.67 GHz was used for the rendering. To minimize thermal throttling issues when collecting timing data, the CPU was set into a slower constant speed of 1.6GHz. The files were rendered to a CMYK color space defined by the eciCMYK ICC profile (FOGRA53)[1] [12] with a bit depth of 32bits/pixel at 1200dpi resolution. For the Images, Altona Visual and Altona 2.0 files this amounts to approximately 0.57 GB, 1.25 GB and 1.11 GB of data respectively. The eciCMYK color space was selected as the destination color space since it was not present in any of the files, forcing significant color conversions. To avoid timing overhead due to file output, the actual output data was written to the null device. The exact command line used was:

```
./gs   -sDEVICE=bitcmyk -Z: -dMaxBitmap=1
       -dNumRenderingThreads=#
       -r1200 -o /dev/null
       -sOutputICCProfile=eciCMYK.icc
       -dGrayValues=256 -f input_file.pdf
```

where the option -dNumRenderingThreads=# varied using from

[1]The eciCMYK.icc profile was included in Ghostscript's ROM file system at compile time.

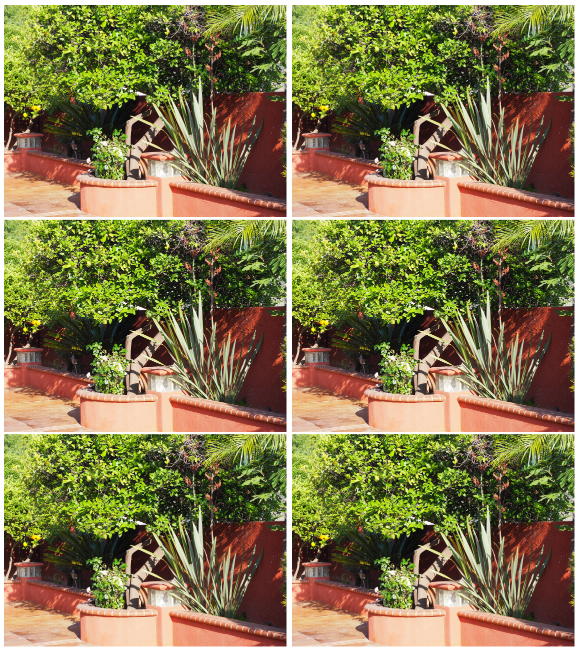

Figure 6. PDF Images document with six images in sRGB color space

1 to 24 threads.[2]

Ghostscript used the contents and size of the file to render each file with a particular number of bands. The Images file was rendered using 168 bands, the Altona Visual file was rendered using 376 bands and the Altona 2.0 file was rendered using 1804 bands.

Using LittleCMS and LittleCMS-MT with Ghostscript, each PDF file was rendered ten times using from 1 to 24 threads. To reduce timing errors caused by the system doing other processes, the minimum rendering time of the ten runs was determined. Figures 9-11 show the minimum timing results (over the 10 runs) for the files comparing LittleCMS to LittleCMS-MT where the x-

[2]The actual numbers used were 0,2-24. The value of 1 has a special meaning in Ghostscript for this parameter and was not used in this testing [2].

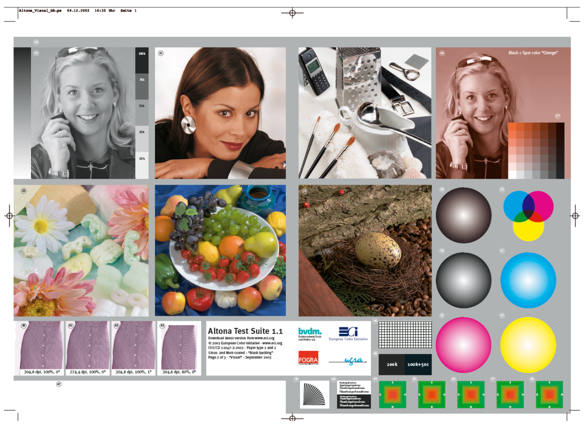

Figure 7. Altona 1.2 Visual test file

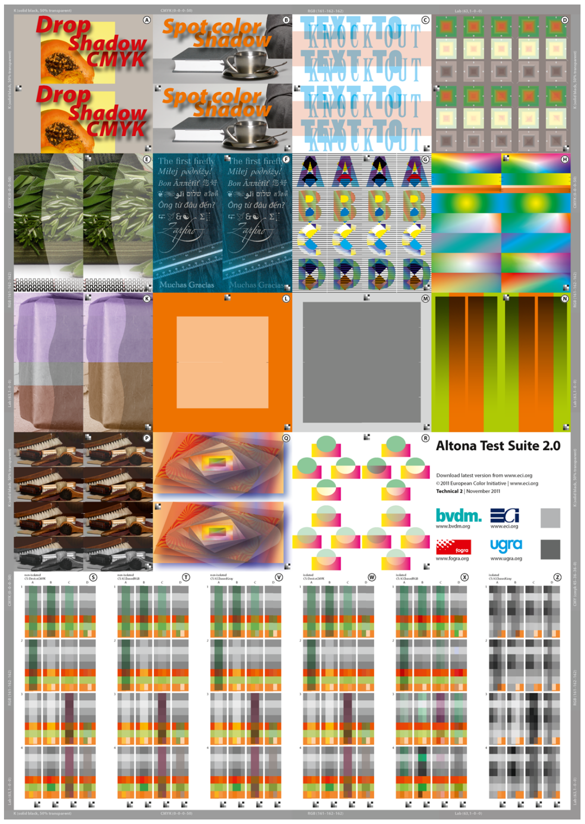

Figure 8. Altona 2.0 test file

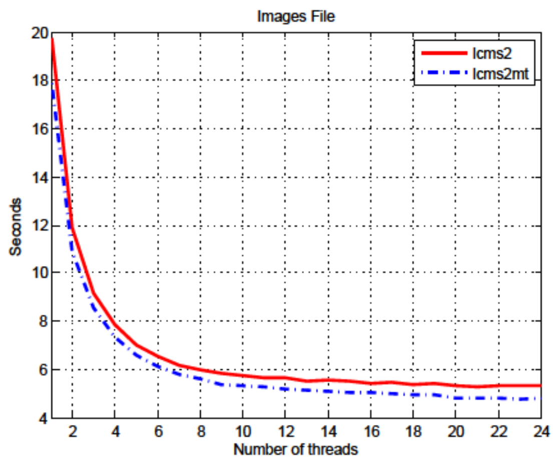

Figure 9. *Timing results for the Images test file*

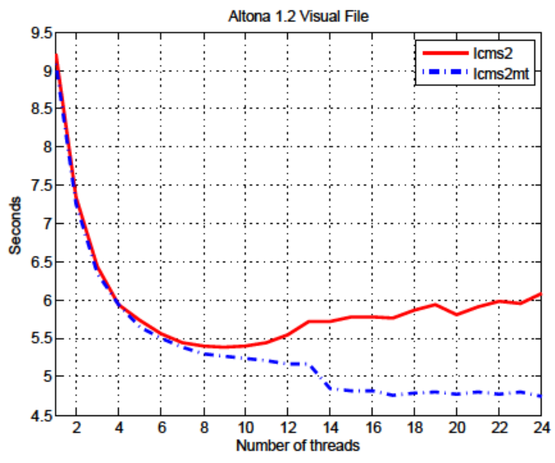

Figure 10. *Timing results for the Altona 1.2 Visual test file*

axis is the number of rendering threads and the y-axis is the time in seconds to render the page. Figures 12-14 show the percent speed up of using LittleCMS-MT vs the number of threads used for each file. The percent speed-up was computed using

$$P = \frac{T_{lcms} \quad T_{lcmsmt}}{T_{lcms}} \tag{1}$$

where T_{lcms} is the time for the page to render using LittleCMS and T_{lcmsmt} is the time for the page to render using LittleCMS-MT. Figures 12-14 are all shown with the same axis range to allow easier comparison.

Discussion

The results show a clear advantage using LittleCMS-MT over LittleCMS for rendering with Ghostscript. Figure 12, which is for the PDF file that contained only images in sRGB color space, showed approximately an 8 percent speed improvement. The source of this speed-up is the reduction of the function call overhead that was discussed in the Pack/unpack section above. For this file, approximately 43 percent of the file processing time

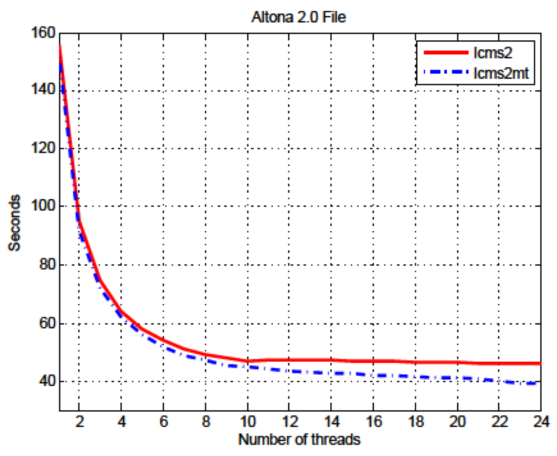

Figure 11. *Timings results for the Altona 2.0 test file*

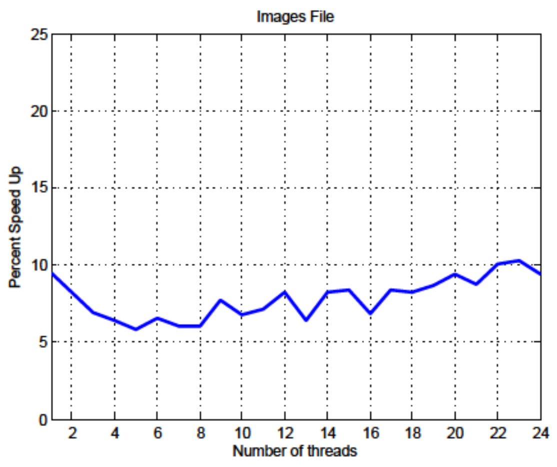

Figure 12. *Percent speed-up results for the Images test file*

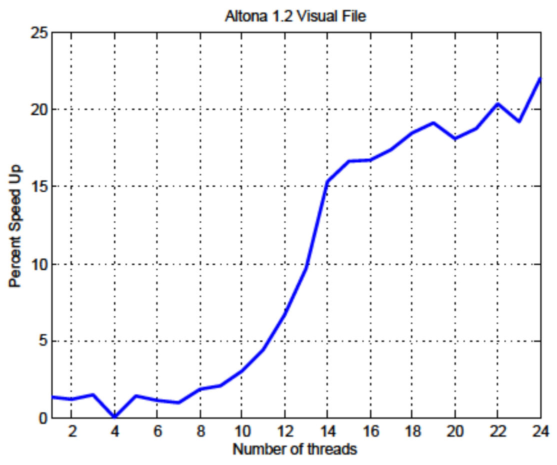

Figure 13. *Percent speed-up results for the Altona 1.2 Visual test file*

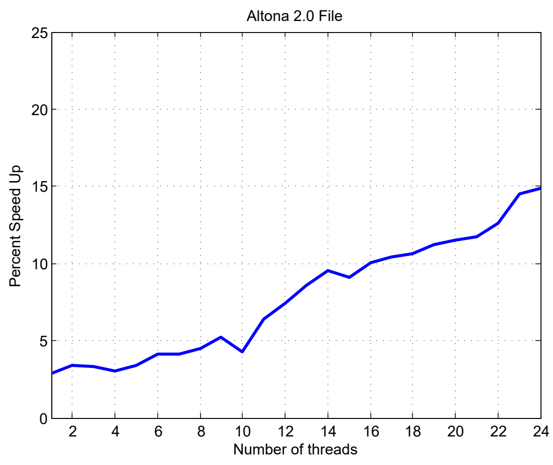

Figure 14. *Percent speed-up results for the Altona 2.0 test file*

was spent in the CMM when running with LittleCMS, indicating that the speed-up within LittleCMS-MT is more like 19 percent. One percent of the processing time was in creating the ICC links while 42 percent was in actually transforming colors. The fact that only one link had to be created for this file is the reason that there is little change in Figure 12 as the number of threads increases.

Figures 13 and 14, which are for the PDF files that contain a large variety of object types and color spaces, show a significant advantage of using LittleCMS-MT as the number of rendering threads increase, with over a 20 percent speed advantage using LittleCMS-MT at 24 threads. The source of this speed-up is the ability of the threads to share links in a common cache and avoid having to each create their own copies or require a lock when they use the links. With the Altona 2.0 file, running with a single thread, approximately 7 percent of the rendering time was spent creating ICC links and 16 percent time was spent transforming colors. With the Altona 1.2 Visual file, approximately 29 percent of the rendering time was spent creating links and 12.5 percent was spent transforming colors for the single threaded case.

Since color conversion is a significant portion of PDL rendering, our work will continue to improve LittleCMS-MT to ensure that it works as efficiently as possible with Ghostscript and MuPDF. With it's liberal MIT license, these improvements should be useful for others who are currently using LittleCMS and faced with multi-threaded bottlenecks or who are transforming significant amounts of data.

References

[1] M. Maria. Little CMS Engine How to use the Engine in Your Application, Available from http://www.littlecms.com (2017)

[2] Source code at http://git.ghostscript.com/ghostpdl.git/. Documentation at https://www.ghostscript.com/Documentation.html (2018)

[3] PostScript® Language Reference, Third Edition, Adobe Systems, Addison-Wesley, Reading Massachusetts (1999)

[4] ISO 32000-1:2008. Document management – Portable document format – Part 1: PDF 1.7 (2008)

[5] ECMA 388 Open XML Paper Specification. Available at http://www.ecma-international.org/publications/standards/Ecma-388.htm (2009)

[6] PCL5 Printer Language Technical Reference Manual Part 1, Hewlett Packard (1992)

[7] Source code at http://git.ghostscript.com/?p=thirdparty-LittleCMS2.git;a=shortlog;h=refs/heads/LittleCMS2-art (2018)

[8] Source code at http://git.ghostscript.com/?p=mupdf.git;a=summary (2018)

[9] Daniel D. Sleator and Robert E. Tarjan, Self-Adjusting Binary Search Trees, Journal of the ACM, vol. 32, no. 3, pp. 652-686 July (1985)

[10] Altona Test Suite 1.2 – Application Kit 16 Reference prints, 7 Color specimens (process color solids), 25 Test suite files, 11 Characterisation data, 11 ICC profiles, documentation, Bundesverband Druk und Medien (bvdm), Wiesbaden, www.altonatestsuite.com (2004)

[11] Altona Test Suite 2.0 - Technical 2, European Color Initiative, www.eci.org, July (2012)

[12] European Color Initiative, profile and details available at http://www.eci.org/en/colourstandards/workingcolorspaces August (2017)

Author Biography

Michael Vrhel was awarded his PhD from North Carolina State University in 1993; during his PhD, he was an Eastman Kodak Fellow. He has many years' experience working in digital imaging, including biomedical imaging and signal processing at NIH; color instrument and color software design at Color Savvy Systems Ltd, and positions at Conexant Systems, TAK Imaging and Artifex Software. A senior member of the IEEE, he has a number of current and pending patents and is the author of numerous papers in the areas of image and signal processing including a book, The Fundamanentals of Digital Imaging. His current interests include efficient computational color rendering methods as well as deep learning applications.