The International Journal of High Performance Computing Applications
0(0):1–13

DOI:doi number
hpc.sagepub.com

Case study of multiple trace transform implementations

Tim Besard* **and Bjorn De Sutter**
CSL-ELIS, Ghent University, Belgium

Andrés Frías-Velázquez† **and Wilfried Philips**
IPI-TELIN/iMinds, Ghent University, Belgium

Abstract

Scientific algorithms are designed and implemented in a variety of programming languages. Depending on the exact application, some languages are a better choice than others: some offer a productive environment while others focus on performance. Selecting a language is often difficult, with poor choices resulting in much higher development times.

By implementing a case study algorithm in multiple programming languages, we compare their pros and cons. As a case study, we selected the trace transform, an image processing algorithm from the widely used class of integral transforms. We describe each implementation, including a highly optimised version for NVIDIA GPUs, and present a productivity overview and an in-depth performance analysis, from which we draw more generic conclusions.

We have found that MATLAB is still the best choice overall, but Julia proves an interesting emerging choice. For realistic images, our CUDA implementation offers the best performance, albeit at a high development cost.

Keywords

Trace transform, MATLAB, MEX, Octave, Scilab, C++, OpenMP, CUDA, Julia

1. Introduction

Modern processors are more capable than ever, allowing for very powerful and dynamic applications. Due to limited advances in single-core performance, more and more of these processors are built on new and innovative multi-core architectures. Such architectures are often incompatible with classical programming models, and sometimes even require specific programming languages.

* Email: Tim.Besard@elis.ugent.be
† Email: Andres.FriasVelazquez@telin.ugent.be

Meanwhile, programmers have been moving towards more productive programming languages, sacrificing performance for reduced initial development time. However, it is then often necessary to reimplement (parts of) the application in a high-performance language to meet performance requirements. This kind of two-tier development method is now often the norm, despite some programming languages trying to offer both productivity and performance.

The balance between performance and productivity is usually very delicate, and the choice for an appropriate programming language depends on many factors. Some specialised programming languages promise great performance, but they are not applicable for all types of applications, and the required expertise can be vast.

To help researchers choose the best programming language, this paper evaluates the implementation of the trace transform in several popular programming languages[1]. This transform is an example of an integral transform; a widely-used class of transforms in medical imaging and in computer vision. In one contribution of this paper, we analyse both the productivity and performance of each implementation and identify issues and aspects relevant for programmers of not only the trace transform, but of many other algorithms as well.

As part of our comparison, we developed an implementation of the trace transform for Graphics Processing Units (GPUs), using NVIDIA's Compute Unified Device Architecture (CUDA) as development platform. As our second contribution, we identified opportunities for parallelism and selected GPU-friendly algorithms to fully utilise the GPU's hardware.

Section 2 starts with an overview of integral transforms, and the definition of the trace transform. In Section 3 we describe the motivation and design of each implementation, followed by an in-depth performance analysis in Section 4. Finally, we discuss these results in Section 5.

2. Integral Transforms

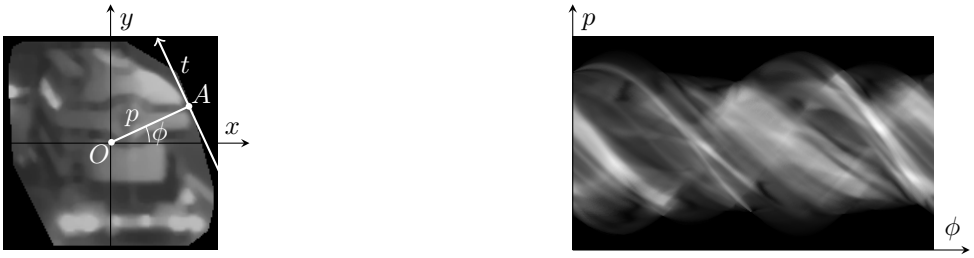
Integral transforms are fundamental operators that transform a function from one domain to another. Examples of these transformations are the Fourier, Laplace, Mellin, and wavelet transforms. They have been extensively used in applied sciences and engineering for solving a large variety of problems. The *integral transform* of a function $f(x)$ with the kernel $K(x, \xi)$ is defined by

$$\mathcal{T}\{f(x)\} = F(\xi) = \int_a^b f(x)K(x,\xi)dx \tag{1}$$

where $f(x)$ is defined in $a \leq x \leq b$, and ξ represents the transform variable. Different kernels $K(x, \xi)$ and values for a and b can be used to derive different transformations.

In image processing, integral transforms play a key role in different primary tasks such as image filtering and pattern matching. For instance, the Fourier and wavelet transforms are fundamental in image denoising and enhancing, while the Mellin and Radon transforms stand out in image recognition. Although the core implementation of an integral transform is carried out with a simple *multiply-accumulate* operation, the actual complexity depends heavily on the image size, data type, and kernel chosen. In the present work, we analyse these and other aspects of the implementation of the *trace transform* [1]. In particular, we consider the trace transform as a case study of integral transforms given its relevance in image processing and its diversity of kernels.

[1] The sources of these implementations are available at `https://github.com/maleadt/tracetransform`



(a) Preprocessed source

(b) Sinogram using the T_4 functional

Figure 1. Source image and sinogram of an image captured from a road tunnel camera.

2.1. Trace Transform

The trace transform is an integral transform that extracts image descriptors by projecting along straight lines of an image in multiple orientations [2]. The transformation originates from the Radon transform, which applies a functional that computes the unweighted sum of the pixel values over each projecting line. By contrast, the trace transform generalizes this concept by proposing functionals based on integrals and medians with different weighting functions. Through careful selection of aggregating functionals, multiple descriptors can be extracted conveying different features of an image [1, 3]. Among these descriptors, the most useful ones are robust to image variations such as affine geometrical distortions, uneven illumination, noise perturbations, and occlusions. Such invariant descriptors have proven useful in a variety of object recognition and identification applications, including face detection [4] and vehicle classification [5].

To introduce the implementation of the trace transform, let $I(x, y)$ be an image with pixel coordinates x and y, as shown in Figure 1a. Moreover, let $x(t; \phi, p)$ and $y(t; \phi, p)$ be the parametric equations of the pixel coordinates on a projecting line in the image, where p is the segment distance between the center of the image O and the closest point A on the projecting line. The parameter ϕ represents the positive angle formed by the x-axis and the segment $\overline{OA}$, while t indicates the pixel position along the projection coordinate. Based on these definitions, we can get the pixel values on a projecting line using $l(t; \phi, p) = I(x(t; \phi, p), y(t; \phi, p))$.

The procedure to extract invariant descriptors from the trace transform is performed in three steps: first, we apply a trace functional, called the T-functional, over the pixel values $l(t; \phi, p)$ of each projecting line. This procedure maps an image from the (x, y) pixel domain into the Hough domain (ϕ, p) yielding a sinogram denoted by $S(\phi, p) = T\{l(t; \phi, p)\}$. For example, processing the image displayed in Figure 1a using the T_4 functional from Table 1 results in the sinogram displayed in Figure 1b. Second, the sinogram is processed by applying a diametrical functional over p, called the P-functional, returning a 1D-signature. This signature is called the *circus function*, and it is defined by $c(\phi) = P\{S(\phi, p)\}$. Third, the circus function is reduced to a single value v, known as the *triple feature*, by applying the so-called Φ-functional: $v = \Phi\{c(\phi)\}$.

Although existing work on the trace transform shows very good recognition results, this comes at the expense of a high computational complexity, limiting its practical use [6]. According to [2], computing the sinogram with n_t samples of the parameter t along the image diagonal, n_p samples of the parameter p, and n_ϕ samples of the parameter ϕ requires approximately $C_T n_t n_p n_\phi / 2$ operations, where C_T is the number of operations per sample of a trace functional. Further computing the circus function requires $C_P n_p n_\phi$ additional operations, where C_P is the number of operations per sample of a P-functional. Finally, the computation of a triple feature requires $C_\Phi n_\phi$ extra operations, where C_Φ is the number of operations per sample of a Φ-functional. By summing up the number of operations of each processing stage, the total computational complexity is approximately $C_T n_t n_p n_\phi / 2 + C_P n_p n_\phi + C_\Phi n_\phi$. As C_T, C_P, and C_Φ are of similar order, we can note

that a large percentage of the complexity comes from the sinogram computation. For this reason, the remainder of this paper will focus on the sinogram calculation.

3. Implementation

When determining the programming languages to realise the different implementations in, we mimicked the typical development flow many algorithms go through;

1. Design and exploration in a high-level, technical computing language.
2. Re-implementation in a low-level language.
3. Optimization for hardware at hand.

For the high-level language, we chose MATLAB [7] as the prime development platform because of its prevalence and perceived ease of use. We also selected Octave [8] and Scilab [9]: two open-source contenders of MATLAB, which should be capable of executing the code developed for the MATLAB implementation.

As a low-level language, we opted for C++, a popular general purpose language which is directly compiled down to native code. This gives us a lot of control over the exact execution, and will prove useful when optimizing for performance. Furthermore, we can use C++ to optimize parts of the MATLAB implementation.

C++ is also a good starting point to optimize for accelerator hardware, because of the good compatibility with libraries and driver toolkits. Using OpenMP [10], we parallelized for multicore processors, and with CUDA [11] we harness the raw power of NVIDIA GPUs.

Additionally, we also created an implementation using the Julia programming language [12]; a relatively new scientific programming language, designed with performance in mind. In the long term, it might even prove a good candidate successor for similar languages such as Python.

Note that this is a fairly limited selection, with some popular languages omitted due to either the required effort of creating another implementation, limited added value or existing experience of the authors.

For most of these languages, the structure of the implementation resembles the pseudocode of Listing 1. This is an alternative way to view the computation of $T\{l(t;\phi,p)\}$, as the cascade of (1) rotating the image over ϕ degrees and (2) aggregating the values of the rotated image column wise using T.

We used a limited number of T-functionals that generate signatures with low information redundancy [3]. This ensures that useful info is summarized in a small number of features, which facilitates further analysis (e.g. object classification). The functionals are listed in Table 1, where $\underset{x\in X}{\mathrm{wmedian}}\,(f(x),w(x))$ represents the weighted median of the function $f(x)$ in the domain X, with weighting or replicating function $w(x)$. All of these functionals use the weighted median to determine the range of elements to be processed by the functional: starting at the median element until the end of each column. In order to find the weighted median, we need to process all elements of $f(x)$ twice; first calculate the total weight based on $w(x)$, and secondly find the element where the running total weight overtakes 50% of the total weight. This means that each functional will need at least three passes through the data, but some functionals require even more. For example functionals T_6 and T_7, which return a second-order weighted median rather than an integral. Worse, that median only operates on strictly positive elements, which means we will have to sort each column in order to get a contiguous slice of valid items for further processing. Note that for other functionals, we can simplify the integrals with summations over pixels, as we only process discrete images.

```
preprocess(image)

# outer loop
for a = 0:angular_resolution:360
    input = rotate(image, a)

    # inner loop
    for x = 1:cols(input)
        column = input[:, x]
        sinogram[x, a] =
            tfunctional(column)
```

Listing 1. Pseudocode of the sinogram calculation.

$$
\begin{aligned}
T_1\{l\} &= \int_{t_1}^{\infty} (t-t_1)\, l(t-t_1)dt \\
T_2\{l\} &= \int_{t_1}^{\infty} (t-t_1)^2\, l(t-t_1)dt \\
T_3\{l\} &= \int_{t_2}^{\infty} (t-t_2)\, e^{i5log(t-t_2)}\, l(t-t_2)dt \\
T_4\{l\} &= \int_{t_2}^{\infty} e^{i3log(t-t_2)}\, l(t-t_2)dt \\
T_5\{l\} &= \int_{t_2}^{\infty} \sqrt{(t-t_2)}\, e^{i4log(t-t_2)}\, l(t-t_2)dt \\
T_6\{l\} &= \operatorname*{wmedian}_{t \geq t_2} \left((t-t_2)\, l(t-t_2), \sqrt{l(t-t_2)} \right) \\
T_7\{l\} &= \operatorname*{wmedian}_{t \geq t_1} \left(l(t-t_1), \sqrt{l(t-t_1)} \right)
\end{aligned}
$$

With: $t_1 = \operatorname*{wmedian}_{t}(t, l(t))$

$t_2 = \operatorname*{wmedian}_{t}(t, \sqrt{l(t)})$

[a] For notational simplicity, $l(t;\phi,p)$ is denoted by $l(t)$

[b] $\operatorname*{wmedian}_{x \in X}(f(x), w(x))$ denotes the weighted median of $f(x)$ in domain X, with weighting function $w(x)$.

Table 1. List of T-functionals.

3.1. MATLAB

MATLAB is one of the most widely used programming languages for technical computing, mainly intended for numerical computation but also usable for a variety of other tasks such as symbolic computation or system modelling. The MATLAB syntax is geared towards matrix manipulation: scalars are treated as 1 by 1 matrices, arrays are represented as a single row or column in matrix, and most parts of the standard library accepts input arguments of a higher dimension, in which case the computation is repeated accordingly. In other cases, the behaviour differs based on the input dimensionality.

MATLAB's *vector expressions* allow for very concise notation, staying close to the original mathematical formulations. For instance, rather than computing $T\{l\}$ for each slice, we compute $T\{l(t;\phi,p)\}$ for all ϕ, p simultaneously by using the necessary MATLAB array operations internally. Apart from the syntactical difference, in many cases vector expressions also greatly improve MATLAB's performance.

Similar to the vector expressions in the standard library, we implemented the trace transform in terms of maximally vectorized operations where possible. Concretely, this means that after rotating we can pass an entire image to the function calculating the T-functional, resulting in an array of values directly corresponding with a single column in the sinogram. By using such coarse operations, we greatly reduced the interpretation overhead generally associated with MATLAB code.

A common optimization for MATLAB applications is to rewrite hot code in MATLAB Executable (MEX) files. These are MATLAB-callable C programs, making use of the MEX API to facilitate compatibility between both languages. This is a typical example of a two-tier architecture (also dubbed as "the two language problem" [12]), where application logic is expressed in a high-level language but heavy computations are performed in a low-level language. In the case of the trace transform, this involved rewriting the sinogram calculation, including all the T-functionals.

However, switching back and forth between compiled MEX and interpreted MATLAB code is costly, significantly lowering performance. In order to avoid this cost we also had to reimplement many auxiliary functions in MEX, duplicating parts of the MATLAB standard library. As illustrated in Table 3, this more than doubled the amount of code. A significant part of these additions can be attributed to boilerplate: verbose and repetitive snippets of code, such as the type conversions needed when calling MATLAB functions or returning values.

Apart from manually converting MATLAB code to MEX, it is also possible to automate (part of) this process. One tool for this used to be the MATLAB Compiler, which compiles source code down to an intermediate

representation and subsequently converts it to MEX code. However, now that MATLAB ships a Just-In-Time (JIT) compiler, this final conversion doesn't improve performance anymore. This is why the MATLAB Compiler no longer supports emitting MEX code, and compiled binaries use the JIT compiler to execute the intermediate representation. The compiler is now only used to deploy MATLAB applications, or to protect their source code.

Another optimization possibility is to use MATLAB Coder, which generates C and C++ code from MATLAB code. This is interesting when deploying MATLAB applications on platforms where MATLAB is not available or difficult to use. The caveat is that the source code needs to adhere to a restricted subset of MATLAB, called Embedded MATLAB. For example, dynamic variables are not supported, and matrix sizes need to be known beforehand. As documented in the user guide, many standard library functions are also more limited when used in context of Embedded MATLAB. Since this is a major effort, and gains would be limited since MATLAB Coder is not meant as an optimizing compiler but rather as a portability tool, we decided not to explore this possibility.

3.2. Octave

The Octave language is a high-level programming language, primarily intended for numerical computations. The language is designed to be source compatible with MATLAB, enabling easy reuse of existing code. A major limitation however is the lack of binary compatibility. This prevents the reuse of MATLAB toolboxes containing precompiled code. Although Octave offers some toolboxes of its own, the degree of interface compatibility with their MATLAB counterparts varies widely.

In the case of the trace transform, only minimal changes were required for the MATLAB implementation to run using the Octave interpreter. In fact, the only modifications related to the use of the image processing toolbox, where extra arguments were required to ensure MATLAB compatibility.

3.3. Scilab

Another popular open-source numerical programming language is Scilab, providing a extensive computing environment for engineering and scientific applications. The Scilab syntax heavily resembles that of MATLAB, but offers less compatibility than Octave does. For example, many standard library functions require slightly different parameters, resulting in wrong computations when code is not properly ported.

When porting the MATLAB implementation of the trace transform to Scilab, quite some syntactical adjustments were necessary. Although Scilab comes with an automated tool for this job, it proved insufficient. Next to these superficial changes, we also had to reimplemented missing functionality. For example, although there exists multiple image processing toolboxes for Scilab, none of them contained all necessary functionality, and combining toolboxes is tricky due to name collisions. We decided to use the Scilab Image and Video Processing (SIVP) toolbox, which seemed more complete than other image processing toolboxes, lacking only image rotation. Lastly, we needed to alter the usage of certain standard library routines, where the behaviour didn't match their MATLAB counterpart. One example of such divergent behaviour is the `sum` routine. When given a 2-dimensional matrix, MATLAB computes individual sums of the elements in each row, while Scilab still returns a single sum of all elements in the matrix. Semantics incompatibilities such as these make it much more difficult to port code, as error-free execution does not guarantee valid results.

3.4. C++

As a second major language, we opted for C++. Not only does this pave the way for further optimizations, it also simplifies usage and deployment on platforms where MATLAB is not available or difficult to use. For example, the most recent MATLAB Compiler Runtime (MCR) weighs over 500MB, and is only available for x86-based platforms. Other platforms require the use of MATLAB Coder, which can only translate a limited subset of MATLAB to standalone C and C++ code. This consideration especially makes sense for image processing algorithms, which are often deployed on low-cost camera hardware.

Using high-level libraries such as OpenCV [13] and Eigen [14] where possible, we tried to minimize the porting effort. A big part of the remaining code increase compared to MATLAB comes from the relative lack of vector expressions. Although Eigen supports basic matrix and vector arithmetic, including several coefficient-wise operations, they proved insufficient for many T-functionals. For example, the T_1 functional multiplies each element by its index; in Eigen this is impossible to express using vector expressions without generating a temporary array containing all indices. Instead, we used manual loop-based computations, performing the same calculation without any overhead. On the other hand, such loops block the library from instantiating more efficient versions of the array, for example taking storage order into account.

OpenMP. The trace transform algorithm is a so-called embarrassingly parallel algorithm, which means little effort is required to split the problem into multiple parallel tasks. Indeed, if we look at Listing 1 there are no loop-carried dependencies in either the outer or the inner loop. This allows concurrent execution of loop iterations, a task for which OpenMP's work-sharing `parallel for` construct is well suited [15]. Annotating both loops results in rotation as well as tracing of columns being performed concurrently. Due to the fact that output values are written into distinct parts of the sinogram matrix, we also don't need any synchronization to merge the results of individual computations.

CUDA. The coarse-grained parallelism that we exploited using OpenMP is not suitable for lightweight threads such as CUDA's: GPU threads execute in what NVIDA calls Single Instruction, Multiple Threads (SIMT) fashion, meaning multiple threads execute the same instructions in lockstep [16]. Even the inner loop of Listing 1, which traces columns with a functional and thus seemingly executing the same instruction stream, is too coarse grained: common image sizes for these applications range between 100 and 200 pixels wide [2, 3]. Spawning only that many threads would not fully utilize modern GPUs, often supporting over 1000 concurrent threads. Worse, it would open the door for thread divergence in case of data dependent execution: this means that individual threads execute different code, breaking the lockstep premise and wrecking performance. For example, all functionals in Table 1 only process part of the domain determined by the weighted median. Concretely, the number of data elements taken into account varies between columns. Depending on the ensuing computation, this could cause threads to diverge significantly.

Calling for a different way of parallelization, we identified fine grained parallelism within the trace functionals themselves. We can split most functionals into at least 3 passes: (1) compute the weighted sum of all elements in a column, (2) use this information to determine the index of the weighted median, and (3) apply the actual functional to part of the column starting from the weighted median. For each of these passes we had to look for GPU-friendly variants of common algorithms, compatible with the GPU's architecture and memory hierarchy. One aspect of this is the aforementioned thread divergence. Even though some divergence can be optimized away by the compiler, for example using predicated instructions, we have chosen to avoid

this potential loss of performance in exchange for some added implementation complexity. All this is a tedious process, requiring both domain and hardware expertise, making programming for GPUs impractical and expensive.

For example, we can parallelize the summation of a column by cascading pairwise sums, computing the final as well as all partial sums in logarithmic time. This is called the prefix sum, or scan, and is a frequently used parallel primitive. The partial sums will also prove helpful to determine the index of the weighted median. In our implementation, we used a work-efficient parallel algorithm avoiding duplicate computations of partial sums, instead communicating with other threads using shared memory [17]. Compared to the naive solution, where each thread is responsible for computing a single partial sum and certain pairwise sums are therefore computed multiple times, this reduces the work complexity from $O(n \log n)$ to $O(n)$ and maps nicely onto the GPU's data-parallel hardware. Note that further optimization is possible, for example by taking physical memory layout into account and reordering memory accesses in order to avoid bank conflicts.

Using the prefix sum information, we can easily determine the index of the weighted median. This happens in constant time, by spawning a single thread for each element in a column. Every thread will check whether its partial sum crosses the median boundary, writing back the local column index if that is the case.

Finally, we calculate the actual functional. This involves processing all elements with a kernel function, and summing the results. Since addition is associative, and the kernel functions only depend on the value being processed, we can parallelize this step in the same way as the prefix sum calculation.

Summarizing, we can implement functionals T_1 to T_5 using 3 distinct kernels;

1. Calculate the prefix sum.
2. Identify the index of the weighted median.
3. Integrate the domain starting from the weighted median using some functional.

For functionals T_6 and T_7, more work is needed. Both require sorting the domain starting from the weighted median, in order to extract values greater or equal to 0. Sorting can be performed in many ways: both serial and parallel, with each a multitude of possible approaches. In the case of a GPU, we need a parallel solution, and one that avoids thread divergence at that. For each thread to execute the same instructions, the sorting should be data-independent. A sorting network fits this description, comparing a set of predetermined data elements at each iteration and swapping if necessary. Apart from the swapping, this is fully data-independent: given the same number of input elements, the number of iterations as well as the comparisons executed by each thread will be exactly the same across invocations. For this application, we use bitonic merge sort, a construction method for building a sorting network [18]. Although work inefficient, especially on partially sorted inputs, the resulting network performs well for small data sizes such as ours [19].

Finally, we also implemented image rotation on the GPU. Given the small image sizes and hence limited optimization opportunities, we opted for a simple implementation, each thread bilinearly interpolating a single pixel. This has several advantages: not only is it significantly faster than rotation on the CPU, after uploading the unrotated source image to the GPU – a relatively costly operation – no further host-to-device transfers are needed any more.

As expected, all of this considerably increases the amount of code. Implementing the functionals required a lot of effort; not only are the computations more complex as described above, most auxiliary libraries such as Eigen are also incompatible with CUDA. Apart from that there is a significant amount of boilerplate required to interface with the GPU and its driver. Worse, the implementation is completely non-portable: besides the fact that the kernels are written in CUDA and thus effectively tied to NVIDIA hardware, they are optimized for current generation GPUs as well. For example, we rely on the maximum number of threads within a block

Processor	Intel i7-3770K (3.5 GHz, 4 cores, 8 threads)
Memory	16 GB DDR3
GPU	NVIDIA GeForce GTX Titan
OS	Debian GNU/Linux 3.14, 64 bit

Table 2. Specifications of the test system.

	Lines of code	
Programming language	**Total**	**Algorithm**
MATLAB	375	200
Octave	375	200
Scilab	500	275
Julia	750	250
MATLAB & MEX	900	650
C++	1250	375
C++ & OpenMP	1260	385
C++ & CUDA	2375	1000

Table 3. Lines of code of entire application and core algorithm for different implementations.

being greater than the vertical resolution of an input image; this might not be the case with future GPUs, and does not even hold true for all current-generation devices. Running the same code on such hardware might result in degraded performance, or even in a failure to launch. Vice versa, when future GPUs become more powerful, the implemented kernels might not fully utilise all available hardware resources. In the case of the GPU used in this comparison, the hardware was already fully utilised without this optimization.

3.5. Julia

As a new contender in the segment of technical computing languages, Julia aims to be a dynamic language for scientific computing, designed for performance through modern language techniques. By means of aggressive code specialization against run-time types in combination with a JIT compiler using the Low Level Virtual Machine (LLVM) compiler framework [20], the compiler is able to emit highly efficient machine code. This makes it possible to write performance-sensitive code in Julia itself, avoiding the classical two-tier architecture. To illustrate this point, the entire Julia standard library is written in Julia itself (with some obvious exceptions for the purpose of reusing existing libraries), while still offering good performance.

Given its similarity to MATLAB, porting the existing implementation to Julia should be a minor effort. However, quite some standard library functions readily found in MATLAB (e.g. `imrotate`, `trapz`, `zscore`, ...) turned out to be missing, more than was the case with Scilab and Octave. Furthermore, in its current state the compiler does not cope well with vector expressions: temporary return values, which only serve as arguments to subsequent vector expression, are not optimized away. This means that verbose, loop-based computations are preferred over their vectorized counterpart. Both of these limitations explain the increases in lines of code seen in Table 3.

4. Performance analysis

We benchmarked the different implementations on a powerful consumer-grade machine, as described in Table 2. For the MATLAB implementations, we used release 2012b. This version of MATLAB ships with version 10.3 of Intel's Math Kernel Library (MKL), and uses the GNU C Compiler (GCC) version 4.4 to compile MEX files. All other C++-based implementations were compiled with GCC version 4.8, using the `-O3` optimization flag. CUDA-code was compiled against version 5.5 of the toolkit, using the official NVIDIA drivers version 340. In the case of Julia code, we used a prerelease of version 0.3 in conjunction with the latest versions of any third-party package as of 2014-07-02. For both Octave and Scilab we used the latest version at hand, respectively version 3.8.1 and version 5.5.

When running the algorithm, we used a realistic source image of size 150x150. This image, shown in Figure 1a, was captured from a road tunnel camera and preprocessed in order to extract the foreground. In case of parallel execution, we relied on the application selecting the optional amount of threads (for OpenMP-annotated code, the runtime library used by GCC defaults to using all 8 logical cores).

Every time, we calculated sinograms for all of the T-functionals listed in table Table 1, running the script or binary multiple times and measuring the execution time of each iteration from within the application. Given measurements in the form of

$$(\text{measured } x) = x_{best} \pm \delta x \tag{2}$$

we estimate the precision of the measurements by calculating the relative uncertainty, defined as the absolute uncertainty divided by the absolute value of the best measurement [21]

$$\text{relative uncertainty} = \frac{\delta x}{|x_{best}|} \tag{3}$$

In our case, the absolute uncertainty and best measurement correspond with respectively the standard deviation and mean of a lognormal distribution [22, 23], estimated using maximum-likelihood fitting. It is generally accepted that relative uncertainties below 2% are characteristic of reasonably careful measurements [21]. For most implementations, 5 measurements proved enough to get a sufficiently accurate result, this after discarding a certain number of warm-up executions. For the CUDA implementation, we iterated 10 times more often, because a steady increase of performance is noticeable across the initial tens of iterations. Discarding the warm-up iterations is not unrealistic, because most applications of this algorithm would be long-running (e.g. processing video material, or calculating many signatures in order to improve recognition). The values reported in the charts and tables below are the means of the aforementioned lognormal distribution, with the maximum relative uncertainty listed in the corresponding captions.

4.1. Overall performance

Comparing the performance of each of the implementations described in Section 3, we can clearly see how our optimization efforts pay off. Both the OpenMP and CUDA versions significantly outperform their peers. Figure 2 shows for each implementation the performance improvement or decline of every T-functional relative to its MATLAB implementation.

Meanwhile, the overhead caused by the MEX conversion layer seems minimal compared to the C++ implementation. Figure 3 zooms in on the execution times of different types of functionals (the performance of functional T_1 closely matches that of T_2, and the same mutually applies for functionals T_3, T_4 and T_5), and shows how the performance of the MEX implementation closely matches that of the C++ version. The performance characteristics of individual functionals will be discussed in Section 4.3.

In the case of pure MATLAB code, execution times depend on the number of available cores. On multicore processors, BLAS operations [24] are parallelized thanks to Intel's MKL. In addition, more elementary (for example trigonometric) functions on arrays are taken care of by Intel's Vector Math Library (VML). Usage of these libraries however doesn't always improve performance: with simple functionals such as T_1 the cost of multithreaded execution actually lowers performance. As task complexity increases (by selecting a more complicated functional such as T_3), or the amount of data to be processed increases (by providing larger images), automatic parallelization pays off again.

Where MATLAB uses Intel's MKL, Julia implements linear algebra primitives [25] using OpenBLAS, which is not optimized as thoroughly as Intel's library. For example, OpenBLAS does not provide different kernels

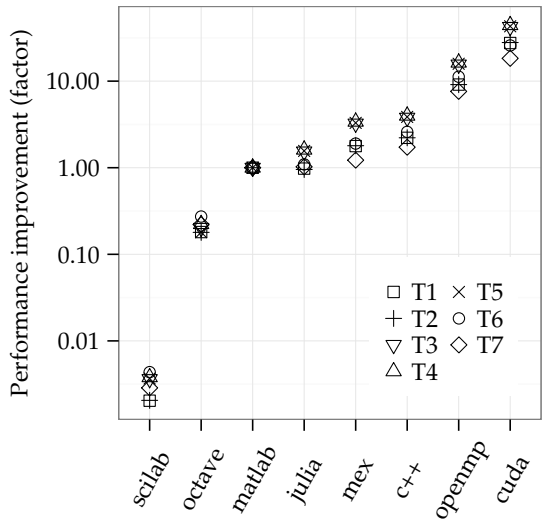


Figure 2. Performance improvement of different implementations over the MATLAB implementation (relative uncertainty: 1.12%).

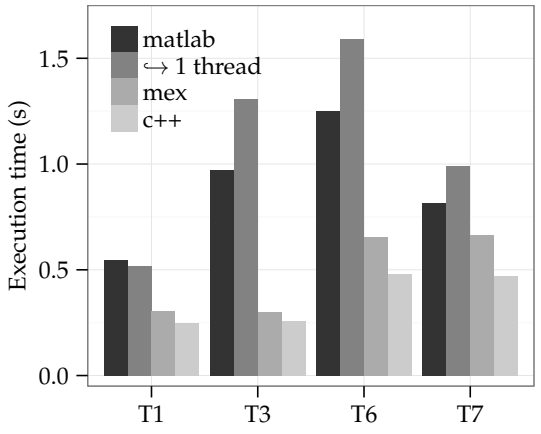


Figure 3. Execution time of MATLAB and C++ implementations on different types of functionals (relative uncertainty: 0.92%).

specialized for various matrix sizes, which makes it underperform at relatively small problem sizes as occurring in the trace transform. Furthermore, the Julia standard library does not attempt to parallelize vector expressions, and its math library doesn't provide functionality for vectorized math. Despite all this, the Julia implementation performs similar to the MATLAB version.

However, one of the key features of Julia is "good performance, approaching that of statically-compiled languages like C".[2] That level of performance is not yet achieved, but if we look at the generated code it seems like performance deficiencies are caused by minor compiler limitations, and C-like performance is very much possible. Concretely, the major loss in performance of this implementation is caused by the bilinear interpolation routine, which is part of the image rotation functionality. As is standard in Julia, each conversion from floating point to integer is checked for exactness, and each access to an array is bounds-checked. But, in the case of the bilinear interpolation routine, most of these checks are superfluous, and if the compiler were smart enough it could omit them. Given time, either LLVM or the Julia compiler are likely to pick up such optimizations.[3] In lieu thereof, we selectively disabled bounds-checking where it proved computationally expensive.

When comparing the performance of Octave and Scilab with their proprietary twin, it is remarkable how well-optimized MATLAB is; both open-source projects perform significantly worse. This is partly explained by the code being interpreted; as mentioned before neither Octave nor Scilab currently ship a JIT compiler, while MATLAB has had one for multiple years already. Also contributing to the inferior performance is the relative lack of built-ins: Octave and Scilab implement most standard library functions in their respective programming language, while in the case of MATLAB many of those are built-in, written and manually optimized in a low-level language such as C++. This also explains the performance difference between Scilab and Octave: in the case of Octave, image rotation is implemented in C++, while in Scilab there was no existing image rotation functionality and we implemented our own version written in Scilab itself. Lastly, MATLAB accelerates many

[2] `http://julialang.org/#A.Summary.of.Features`
[3] `https://github.com/JuliaLang/julia/issues/3440`

vector operations using Intel's VML libraries as mentioned before, which is not the case for either Scilab or Octave.

4.2. Image size

Figure 4 compares the execution time of each implementation used with differently sized input images, calculating all functionals from Table 1. This shows that the performance characteristics are not clear-cut. For example, although MEX performs consistently faster than MATLAB, the gap between both gets smaller as the image size increases. In the case of small images, MEX performs up to 3 times faster, but for large images MEX only performs about twice as fast. This is explained by MATLAB using Intel's MKL and VML libraries. As described before, these libraries parallelize computations when cost-effective. In MEX on the other hand, all calculations are written in plain C, with vector expressions lowered to loops processing each element individually. This avoids the cost of calling into a library and selecting an appropriate implementation. For small images this constant cost is significant compared to the limited amount of work which has to be done. For larger images, not only does this cost contribute less to the total execution time, optimizations such as parallel execution are now cost effective and allow improving the performance.

Next to the input-dependant scaling of the execution time, some implementations suffer from a constant overhead, illustrated by the deviation from a straight line in Figure 4. Regarding the MATLAB, MEX and Octave implementations this is due to the cost of interpreting source code which does not depend on the input image size. Although MATLAB has been using a JIT compiler for a while [26], it does not fully lower the source to machine code. Similarly, the MEX version still contains a fair share of MATLAB code, albeit much less. In the case of Octave, all code is still fully interpreted, although a JIT compiler is being worked on.[4] Meanwhile, Julia fully JIT-compiles all source code, avoiding such a constant overhead. The only difference with a fully native implementation such as the C++ version is the lower quality of generated code, as explained before. Note that Scilab too interprets all code, but the cost thereof is insignificant and hidden by the logarithmic scale.

In the CUDA implementation, there is a much more significant constant overhead, which can be accredited to the cost of configuring the GPU and launching the kernels. Since the number of kernel launches is a function of the angular resolution and the requested set of T-functionals, it is constant in terms of the input image size. By asynchronously issuing kernels and overlapping the next launch with the actual execution of previous ones, we minimize the practical cost. But for small images this does not help, as the kernels finish faster than the time it takes to enqueue the next launch.

Lastly, the OpenMP implementation exhibits no perceptible constant overhead, which is unsurprising given the coarse level of parallelism.

4.3. T-functionals

Different T-functionals exhibit different performance characteristics. Figure 5 visualizes for each implementation the execution time of every T-functional, normalized against T_6 which usually is the most computationally-intensive functional. Note that this normalization is performed for each implementation individually. As a consequence, this chart cannot be used to compare cost values of different implementations. For example, in the case of T_1 the relative execution time of MEX is greater than that of MATLAB; this does not mean MATLAB is faster, but merely that the relative cost of the T_1 functional is greater when calculated with the MEX implementation than when calculated using MATLAB.

[4] https://www.gnu.org/software/octave/doc/interpreter/JIT-Compiler.html

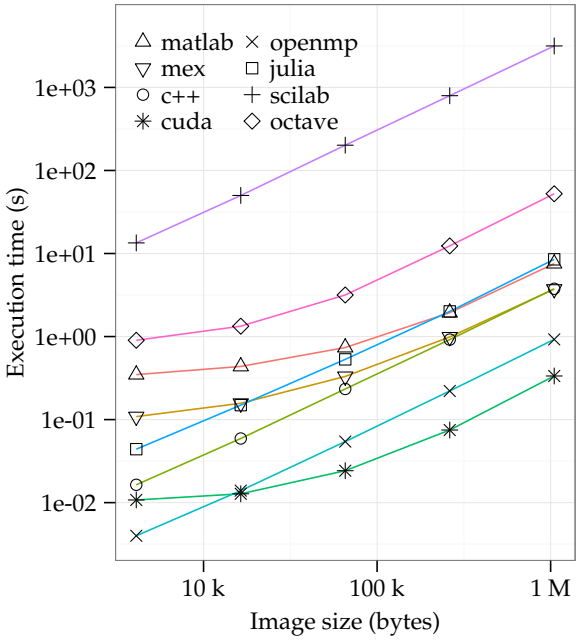


Figure 4. Impact of image size on execution time of different implementations (relative uncertainty: 1.65%).

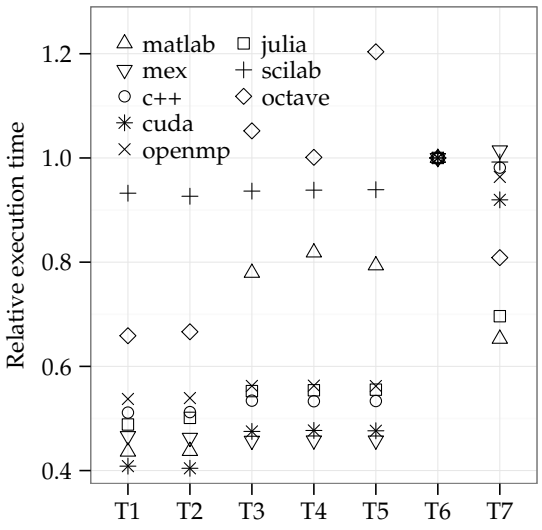


Figure 5. Execution time of different T-functionals, normalized against T_6 (relative uncertainty: 1.12%).

As expected from the definitions in Table 1, functionals T_1 and T_2 are the cheapest to compute because they compute a fairly simple integral.

Similarly, functionals T_3, T_4 and T_5 are grouped because of their similarity. For most implementations, the cost of the complex integration weighs significantly on the execution time. Notable exception to this is the MEX implementation, where the execution time of functionals T_1 through T_5 is almost identical. Even though the actual kernel for these functionals is more complex (i.e. summing over an extra variable for the imaginary part, and a call to `hypot` to get the absolute value), the resulting execution time only increases minimally. Because MEX binaries are notoriously hard to profile or simulate, especially when not easily isolated from the rest of the application, we could not find a reason for this. In the case of MATLAB, the execution time increases significantly. This is because these functionals do not map as closely to array expressions, resulting in multiple fine-grained calls to the standard library. Without a JIT, the effect could be even worse: since most of the computations in these functionals are element-wise computations, the interpreter would have to process loops iterating over each element individually. This explains why both Scilab and Octave are significantly slower than MATLAB, as shown in Figure 2. Interestingly, while Scilab's performance is low but identical for all three functionals, the execution time of the Octave implementation varies. This is explained by looking at the actual expressions being integrated in these functionals. Concretely, functional T_4 integrates $f(s)$, while T_3 integrates $sf(s)$ and T_5 further raises complexity by integrating $\sqrt{s}f(s)$. This translates to respectively 1, 2 or 3 element-wise operations, implemented as loops processing each element of every image. Octave does not manage to combine these loops, resulting in multiple independent passes over the entire image. Other implementations (such as C++ and CUDA) do not suffer from this issue, because due to the lack of array expressions we manually expanded these computations into loops, combining computations along the way.

Lastly, the T_6 and T_7 functionals both perform significantly slower because of the multiple required passes through the data. Of these two functionals, T_6 seems like most computationally expensive one because the

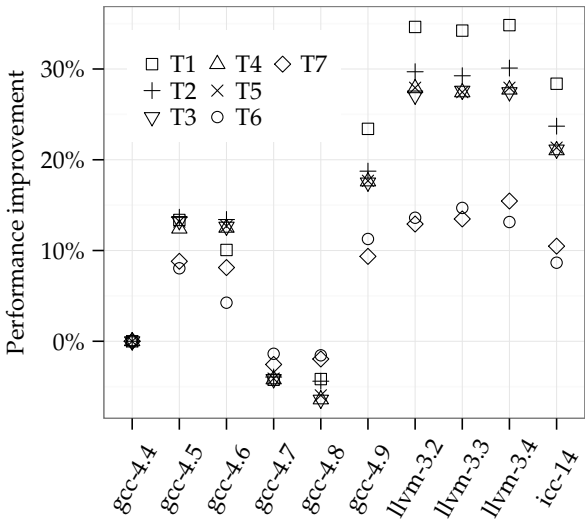


Figure 6. Performance improvement with different C++ compilers, relative to GCC 4.4 (relative uncertainty: 0.48%).

input values also need to be permuted before calculating the median. However, since T_6 uses the squared weighted median, fewer elements are being processed compared to T_7. The resulting performance depends on both of these factors, and differs between implementations. In the case of MATLAB, Julia and Octave, processing fewer elements does not weight up against the added cost of permuting those elements; hence the execution time of functional T_6 is significantly higher compared to T_7. Although we'd expect the same for Scilab, the slowdown is minimal; this is because the time needed to calculate the functional is dwarfed by the overhead of calling the actual function implementing each functional. At the other end of the spectrum, C++ and MEX perform mostly identical on functionals T_6 and T_7. Although the same seems to apply for CUDA, this implementation does not process any fewer elements, but processes entire images at once and masks unnecessary operations to elements before the weighted median in order to avoid divergent execution.

4.4. C++ compiler

When compiling C++ code, multiple parts of the toolchain can influence the performance of the generated code. Most noticeably, there is the compiler, impacting performance by generating better or worse code. However, there is also the standard library, providing the programmer and the compiler with implementations for commonly used functionality. The quality of these implementations can vary greatly. For example, older versions of the GNU C Library (glibc) implemented the call to `sincosf`, which calculates both the sine and cosine of a floating-point number at the same time, using two separate calls to both `sinf` and `cosf`. Starting with version 2.15, a platform-optimized version for `x86_64` processors was added, avoiding these separate calls and calculating both values at the same time using SSE2 vector instructions. Since we actively use this library function when rotating the source image as well as in functionals T_3 to T_5, performance improved greatly when using more recent versions of glibc.Because these differences are mostly independent of the compiler, we decided to use a fixed version of glibc (version 2.19, the latest at the time) for all measurements.

Much to our surprise, even after aligning standard library usage there were still significant differences in execution times depending on which toolchain the C++ implementation was compiled with. Figure 6 shows the impact of using a certain compiler on the execution time of each T-functional, relative to the execution time of that T-functional when compiled with GCC 4.4. Unless stated otherwise, the analyses below will only

compare the performances of the T_1 functional. Even though the performance improvement varies between individual T-functionals, this mostly depends on the library usage and not on the compiler. For example, the T_1 functionals relies on almost no external functionality, which means practically all executed instructions are generated by the compiler under investigation. In other cases, most notably T_6 and T_7, functionals rely more heavily on library functions. The compiler cannot change these functions, which means it can has less of an impact on the total execution time. For the remaining performance differences, the rationale tends to be the same for all functionals, scaling uniformly between compilers.

In the following subsections we will analyse the behaviour of binaries generated by different compilers. In order to get an approximation of time spent in each function, we used the gperftools CPU profiler [27]. To get a more complete view of how the application behaves, we executed the binaries using callgrind, a call-graph generating cache and branch prediction profiler part of the valgrind tool suite [28]. This yielded an instruction-accurate trace of the program execution, but didn't prove accurate enough to estimate the actual cost of execution. We used the Linux perf tools to record hardware performance counters [29], providing us with accurate measurements of those metrics as well.

GCC. For example, when switching from GCC 4.4 to 4.5, the billinear interpolation function is not longer inlined into the image rotation routine any more. Although this results in 10% more instructions, hardware performance counters reveal that the total number of cycles drops by 11% due to fewer instruction cache misses, making for for a 24% higher Instructions Per Cycle (IPC) ratio.

If we upgrade to either GCC 4.7 or 4.8, performance degrades again. But this time there is no obvious culprit: comparing to binaries compiled with GCC 4.6, the number of executed instructions even drops slightly, and inlining behaviour has not changed, yet the IPC is lowered by 18%. Looking closer into the hardware performance counters, this is again due to cache behaviour: 12% more instruction cache misses make for 61% more front-end stalls. Probable cause for this is the compiler laying out instructions in a more cache-unfriendly manner.

GCC 4.9 fixes this regression, and further improves performance by raising the IPC by 13% compared to GCC 4.6. This is mostly caused by a 16% fewer instruction cache misses, reducing front-end stalls by 23%. Although these performance improvements are significant, the generated code is again hardly different. Compared to the binary generated by GCC 4.8, the code is laid out only slightly differently, reducing the number of branches in hot parts of the application.

LLVM. Looking at the LLVM-based Clang compiler, all of the tested versions yield similarly performing binaries. Interestingly, they consistently outperform versions compiled by GCC. Analysing the generated code, much fewer instructions are executed: about 17% less than with GCC 4.9. This negates the apparent 9% lower IPC, ultimately resulting in 8% fewer cycles executed. There are multiple reasons for the smaller code size. Just like GCC 4.4, all versions of LLVM inline the billinear interpolation function into its only call site. This allows for specific optimizations, and removal of unnecessary code (such as argument passing and register saving). But more importantly, LLVM frequently generates more specific instructions tailored for modern processors. The choice between these instructions is based on the compiler's cost model for different processor families, modelling the latency of instructions and usage of processor resources. It seems that in the case of the Intel Ivy Bridge processor used in this test, the machine model of LLVM helps the compiler to generate better performing code than GCC does.

ICC. We also tested the Intel C Compiler (ICC) version 14, as part of the Intel Composer XE suite. This compiler is renowned for its high quality code generation for Intel processors. ICC ships its own high-performance math library, replacing calls to the system math library and linking it into the final application. But despite all this, the resulting performance is not best in class. Comparing with the fastest binary courtesy of LLVM 3.3, ICC generates code performing slightly worse. Looking at hardware performance counters, the observed behaviour is similar as well. However, if we trace the execution using a simulator, many differences come to light. For example, ICC inlines functions much more aggressively, even when there are multiple call sites. This allows for heavy specialization, but increases the risk of instruction cache misses and enlarges the resulting binary (the binary generated by ICC is twice as big as the next in rank). And indeed, executing the binary generated by ICC causes 50% more instruction cache misses, partly due to some bad inlining decisions.

Intel also ships their MKL and VML libraries together with the ICC compiler, providing optimized mathematical routines for x86-compatible architectures. Eigen specifically supports both MKL and VML, and uses them if available. However, using these libraries actually degrades performance, raising the number of cycles by 5 to 10% (depending on the exact T-functional). This is caused by either Eigen using these libraries in a subpar manner, or the library implementations actually being suboptimal. We could not verify either of these hypotheses, due to MKL containing unusually encoded instructions not supported by valgrind.

4.5. OpenMP compiler

Next to the impact on single-threaded performance, we also evaluated the effect of switching compilers for multi-threaded applications by analysing the performance of the OpenMP implementation. Since LLVM does not officially support OpenMP yet, we had to use a development version based on LLVM 3.4.[5] This version relies on the Intel OpenMP Runtime Library[6], which is also used by the ICC compiler.

In order to quantify the effect of using OpenMP, we calculate the speedup [30] compared to the original implementation

$$S = \frac{T_{old}}{T_{new}} \tag{4}$$

As per Amdahl's law, the expected speedup S is a function of the number of threads of execution n, and the fraction of the application B executed serially

$$S(n) = \frac{1}{B + \frac{1}{n}(1 - B)} \tag{5}$$

In the case of the trace transform, most part of the algorithm is parallelized, and hence the serial fraction is very small. More concretely, even in the worst-case scenario where we process a small image with a lightweight T-functional (resulting in a smaller parallel fraction), the serial part only makes up at most 1% of the total execution time. For realistically-sized images, the serial fraction drops well below 0.1% for all T-functionals.

Normal execution. The processor used in our experiments contains 4 cores, supporting as many concurrent threads. Assuming a serial fraction of 0.1%, Amdahl's law predicts a speedup of 3.99. The actual speedups are visualized in Figure 7, where grey data points indicate measurements using 4 concurrent threads. Impressively, the speedup never falls below 3.8. In the case of GCC 4.8, most functionals are even sped up by a factor of 3.94, which is only 1% away from the theoretical maximum. Using ICC, the speedup even exceeds that theoretical limit, but that is most likely due to suboptimal execution of the single-threaded version.

[5] https://clang-omp.github.io/
[6] https://www.openmprtl.org/

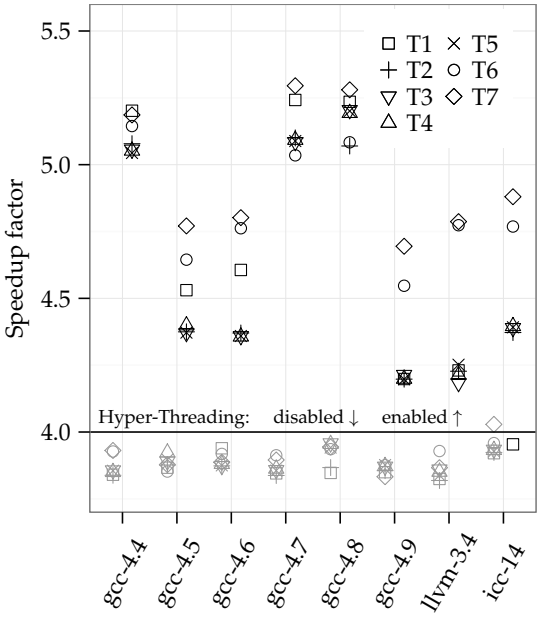


Figure 7. Speedup of using OpenMP with different C++ compilers (relative uncertainty: 0.54%).

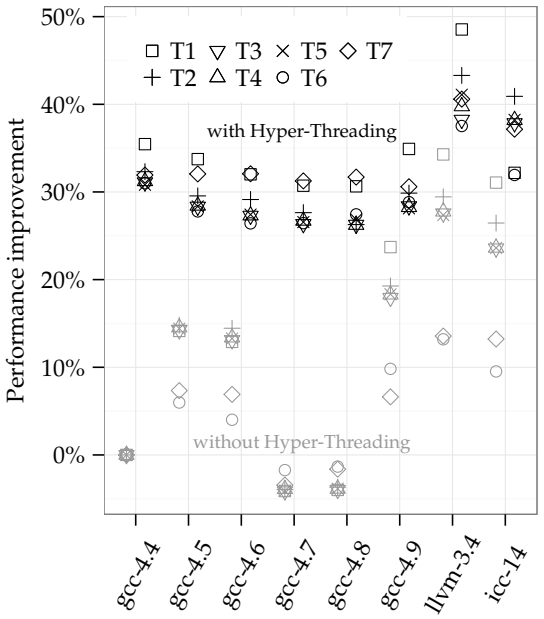


Figure 8. Performance improvement of using OpenMP with different C++ compilers, relative to GCC 4.4 with Hyper-Threading disabled (relative uncertainty: 0.06%).

Overall, all compilers manage to parallelize computations without too much runtime overhead. Slight differences between T-functionals are to be expected; some functionals (such as T_6 and T_7) take longer to compute, increasing the total fraction of the application executed in parallel.

Hyper-Threading. If we enable Intel's Hyper-Threading technology, the processor exposes two logical cores for each physical one. This allows sharing execution resources which would otherwise be idle during execution of a single thread. On paper, this raises the maximal number of concurrently executed threads to 8, which makes for a theoretical speedup of 7.94. In practice however, some execution resources will be exhausted and prevent threads from executing as fast as they would normally, lowering the speedup significantly below the theoretical limit. This is illustrated by the black data points in Figure 7. The speedup now ranges from 3.95 to 5.30, with the mean only at 4.29.

Interestingly, different compilers now yield vastly different speedup factors, with significant divergence between individual T-functionals as well. This performance behaviour is opposite of the single-threaded analysis as shown in Figure 6. The cause for inter-compiler differences is code generation quality: when compiled code efficiently uses much of the processor's execution resources, i.e. without much stalling or idle components, single-threaded performance will excel but there will be few opportunities for Hyper-Threading to improve performance in case of multi-threaded execution. Performance differences between T-functionals can be attributed to similar effects: T_1 is a computationally cheap functional (i.e. doesn't use many execution resources), and both T_6 and T_7 heavily rely on sorting which involves plenty of random memory accesses causing a lot of stalls.

Where Figure 7 plots the speedup of each execution relative to its single-threaded version, Figure 8 compares the execution time to that of a binary compiled with GCC 4.4, executed with Hyper-Threading disabled.

	MATLAB	Scilab	Octave	MEX	C++	OpenMP	CUDA	Julia
Vector expressions	++	+	+	-				−
Use of built-ins	+	++	++	-				=
Compare toolkit versions	+	++	++	+	+	+/=[a]	=	++
Avoid context switching				++		=	+	=
Minimize memory transfers					+	+	++	=
Memory allocations					+	+	++	
Floating-point precision					+	+	++	
Avoid thread divergence						=	++	
Decomposition into parallel subproblems						+	++	
Minimize dependencies within loops/kernels						++	++	
Minimize dependencies between loops/kernels						=	++	
Type stability								++

[a] Hyper-Threading respectively disabled/enabled

Table 4. Difficulties and points of attention for each language in order to obtain good performance.

For executions with Hyper-Threading disabled, shown as grey data points, the trends are identical to the single-threaded versions (correlation coefficient 0.99). Enabling Hyper-Threading, it is clear how the technology manages to compensate poor code generation as explained before, levelling-out performance characteristics where there used to be significant differences in single-threaded performance. However, this also means that the single-threaded performance is not a good predictor for multi-threaded performance any more (correlation coefficient drops to 0.53). Worse, in some cases the ordering of compilers changes: for example, without Hyper-Threading GCC 4.5 clearly outperforms version 4.4, but this changes when enabling the processor feature.

5. Discussion

Depending on the type of application and purpose of the implementation, some programming languages are a better choice than others. As shown in previous sections, many aspects to this are non-obvious and not evident beforehand, yet heavily influence both development time and resulting performance. In the following section, we will try to generalise experience from implementing the trace transform in different languages, and formulate more broadly usable guidelines.

5.1. Design and exploration

When the main purpose is design and exploration, general ease of use is at the top of the list of priorities. This encompasses the learning curve of the language, quality of development tools, availability of auxiliary packages, ... Another important aspect is the performance of unoptimized but relatively idiomatic code, i.e. using language features and syntactical constructs as they are meant to be used. Table 4 lists some considerations to keep in mind in order to obtain good performance.

Looking at MATLAB, it still reigns the field of technical computing. It offers a good, mature development environment, and supports a wide range of toolboxes augmenting the language with domain-specific features. Although many languages surpass MATLAB on certain levels, for example the toolbox system is overshadowed by package manager such as Julia's, and many modern languages offer more consistent and user-friendly syntax, none of these languages manage to offer a platform as comprehensive as MATLAB.

Regarding performance, MATLAB isn't great but dominates most of its direct contenders (i.e. Scilab and Octave). For fine-grained computations such as for loops, this is thanks to its JIT, avoiding the interpretation overhead by lowering source-code to an intermediate representation. But the JIT doesn't compile to native code, which explains why built-in functions written in a lower-level language still offer more performance. MATLAB ships with many of these built-ins, some of which utilising multiple processors when cost-efficient. This means that for large-scale computations, MATLAB can be reasonably fast. In the case of the trace transform, such computations are easy to find and exploit. But for other types of applications, this might not be the case, and MATLAB might perform significantly worse.
Altogether, MATLAB's performance is acceptable, but heavily depends on the type of application and proper usage of the language. Note that the exact set of built-ins can differ between both MATLAB and toolbox versions, so it is advised to track updates to these components as they appear.

In some cases, one of MATLAB's clones can be preferable. For example, both Scilab and Octave are free and open source, and easily deployable on a broader set of platforms than MATLAB is. Scilab seems like the most complete alternative to MATLAB, offering a full-fledged user interface with support for many toolboxes. The performance is disappointing however, mostly due to the absence of a JIT and the relative lack of built-ins.
Octave is another MATLAB clone, paying close attention to source-level compatibility. Similarly to Scilab, it offers a comprehensive set of toolboxes, but currently lacks an integrated user interface (although one is under development, and third-party alternatives exist). Performance tends to be better than Scilab, and a JIT is in development. In its current state however, performance still heavily relies on the use of built-ins.

As expected, low-level languages such as C and C++ are not very useful for explorative programming. Development environments fit for scientific work are mostly non-existent. Their syntax is often much more verbose, and they come with a limited standard library mostly geared towards general purpose programming. Although this can be solved by the use of high-level utility libraries such as Eigen and OpenCV, these are often difficult to glue together and not always easy to install let alone deploy in a robust manner.
Because these languages are designed as a fairly limited abstraction on top of actual hardware, computations are usually modelled as loops processing elements rather than operations performed on a set of elements, as is common in technical languages. This is semantically further away from the original mathematical expressions, making the code less natural and more difficult to comprehend. Current C and C++ compilers are also better at optimizing such loop-based computations, employing advanced heuristics to improve performance. Note that these heuristics change often: different compilers, or even different versions of the same compiler can generate vastly different code.
Although the opportunity for performance is much bigger than with high-level languages, offering way more control to the programmer, naively implemented code will often not perform very well. Where high-level languages detect certain operations and call high-performance libraries instead, in low-level languages the programmer would have to do all of this himself. This especially makes a difference when large inputs are involved.

Julia is an interesting combination of high-level language features designed in such a way that they still map onto efficient native code. Sadly, the project is still in its infancy; despite very promising results it is not yet ready for production work. For example, there is no decent graphical development environment yet, but several are currently in development. Core features (such as a debugger) are often missing, and the language can still change considerably. The standard library and package ecosystem are also quite limited, albeit expanding steadily.
Performance-wise, applications written in Julia generally perform well, as a result of thorough type inference and JIT-compilation of that type-specialised code to native machine instructions. However, it is easy to wreck

performance without the compiler warning about anything. For instance, type conversions are expensive, but, Julia being a dynamic language, it is easy to write type-unstable code causing a lot of type conversions. The compiler also doesn't cope well with vector expressions; loop-based computations are preferred and yield much better performance. Lastly, Julia doesn't use the high-quality math libraries MATLAB has available, such as Intel's MKL and VML. This means that while Julia is intrinsically much faster than MATLAB, for large vectorized inputs MATLAB manages to offload large parts of the computations, and quickly catches up with Julia.

5.2. Optimization and usage of accelerators

When implementing for performance, not only the initial but also the potential performance matters, and how much effort it takes to optimize up to that point. One aspect of this is how easy accelerators, such as GPUs or multicore processors, can be used. Such accelerators are often exposed through low-level libraries, so it is important to have a powerful Foreign Function Interface (FFI).

In the case of MATLAB, the picture is less rosy than before. In its current form, the JIT cannot lower source code to efficient native instructions. Reimplementing hot code in MEX solves this, but quickly ruins productivity both because of MEX's verbosity, and because of the need to reimplement many standard library functions in order to avoid expensive calls from MEX into MATLAB. Worse, using MEX in some cases degrades performance, because MATLAB cannot substitute operations with high-performance alternatives any more.
In order to make use of accelerator hardware, MATLAB provides wrapper toolboxes like the parallel computing toolbox. But these toolboxes are expensive, and in the case of CUDA don't even support the entire array of features. Manually wrapping libraries, either with MEX or using FFI functionality, is cumbersome and verbose.

With Scilab and Octave, the low base performance makes them less interesting for performance-oriented applications. Additionally, both Scilab's and Octave's FFI features are even less developed than MATLAB's, complicating the use of external libraries. For Octave, the popular Simplified Wrapper and Interface Generator (SWIG) tool is available, but this still requires manually creating a SWIG interface file, and the generated wrappers are written in C++ rather than in Octave. Although there already exist some wrapper packages to interface with hardware such as GPUs, most notably for Scilab, they are limited in scope and heavyweight in usage.

Employing C++, flexibility is unparalleled. As most high-performance libraries are written in either C/C++ or Fortran, compatibility comes naturally, without the need for wrapper libraries or expensive type conversions. Most compilers also support OpenMP, which allows to make use of available multithreading capabilities in a very lightweight manner. OpenMP also proved to be very efficient, approaching the theoretical limit of the attainable speedup.
The choice of compiler is an important consideration, significantly influencing both single- and multi-threaded performance. In case of multi-threaded execution, the use of Hyper-Threading can compensate for lower-performing binaries, but the effects are not always straightforward and should be checked on a case-by-case basis.

Using CUDA from C++ is well-supported, but cumbersome: the programmer is required to take care of many technicalities, such as how different memories are managed, or how the GPU's processors are configured and organised. The code executed on the GPU is also low-level, representing what a single thread executes rather than describing the high-level operations performed on data. This requires the programmer to decompose each operation into parallelizable subproblems, find GPU friendly implementation of these problems, and manage dependencies between individual threads of execution. Although there are some libraries offering

parallel building blocks, this only solves part of the problem and often degrades performance due to the lack of global analysis.
What proved effective for performance was to fully move a contiguous set of computations, decomposed into multiple kernels, to the GPU. This made it possible to use the host processor only for kernel launching, keeping the GPU busy and minimizing latencies between kernels, and minimizing expensive memory transfers between the host and the GPU.

Since Julia has been designed with performance in mind, it excels at optimization opportunities. Contributing most to this, is the lightweight yet powerful FFI. Since Julia's JIT generates native code, calls to foreign functions have no added overhead. Additionally, most primitive types map directly onto natively supported ones, avoiding costly type conversions before calling foreign functions. It is even possible to embed inline assembly in Julia source code, allowing for extremely fine-grained control over what gets executed.
Using this functionality, many libraries have already been wrapped as easy-to-use packages. In order to simplify this effort, a wrapper-generator tool exists, using Clang to parse library headers and automatically generate appropriate Julia wrappers.[7]
Despite these capabilities, the hardware we looked at is currently not really usable yet from within Julia. In the case of GPUs, the CUDA packages only interface with the low-level driver, and it is non-trivial to actually run and interface with code running on the device. For multicore processors, parallel primitives exist, but they are heavyweight and far less usable than the annotations in OpenMP. Interfacing with an OpenMP runtime library isn't possible either, because the core infrastructure of Julia is not thread-safe yet.

6. Conclusion

In this paper, we have implemented several versions of the trace transform algorithm using MATLAB (optimized using MEX), Octave, Scilab, C++ (optimized using OpenMP and CUDA) and Julia. For each of these versions, we evaluated the ease of design and implementation, initial performance, and opportunities for optimization.

We have shown that none of the tested programming languages offers both a high-productivity environment, and the potential for performance without resorting to reimplementation in a lower-level language. The best candidate is Julia, but this language is still in its infancy which strongly manifests in terms of development tools.

For each of the listed implementations, we have conducted a thorough performance analysis, and drawn conclusions which can also help to optimize algorithms resembling the structure of integral transforms, and even more generic applications.

Lastly, we have also contributed a high-performance GPU implementation of the trace transform using NVIDIA's CUDA. Except for unusually small images, it outperforms all other implementations, improving the original MATLAB performance with more than one order of magnitude. This can be a base for further, real-time research, previously impractical due to the high computational complexity of the trace transform.

Acknowledgements

This work has been supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT Vlaanderen), and by Ghent University through the Concerted Research Action (BOF-GOA) program on distributed smart cameras.

[7] `https://github.com/ihnorton/Clang.jl`

References

[1] Petrou M, Kadyrov A. Affine invariant features from the trace transform. 26th IEEE Transactions on Pattern Analysis and Machine Intelligence. 2004;26(1):30–44.

[2] Kadyrov A, Petrou M. The trace transform and its applications. 23rd IEEE Transactions on Pattern Analysis and Machine Intelligence. 2001;23(8):811–828.

[3] Frias-Velazquez A, Ortiz C, Pizurica A, Philips W, Cerda G. Object identification by using orthonormal circus functions from the trace transform. In: Proc. 19th International Conference on Image Processing. IEEE; 2012. p. 2153–2156.

[4] Srisuk S, Petrou M, Kurutach W, Kadyrov A. Face authentication using the trace transform. In: Proc. Computer Society Conference on Computer Vision and Pattern Recognition. vol. 1. IEEE; 2003. p. 305–312.

[5] Frías-Velázquez A, Van Hese P, Pižurica A, Philips W. Vehicle classification for road tunnel surveillance. In: IS&T/SPIE Electronic Imaging. SPIE; 2013. p. 1–6.

[6] Fahmy SA, Bouganis CS, Cheung PY, Luk W. Real-time hardware acceleration of the trace transform. Journal of Real-Time Image Processing. 2007;2(4):235–248.

[7] The MathWorks, Inc . MATLAB R2012b; 2012. `http://www.mathworks.com/products/matlab/`.

[8] Octave community. GNU Octave v3.8.1; 2014. `www.gnu.org/software/octave/`.

[9] Scilab Enterprises. Scilab: Free and Open Source software for numerical computation; 2012. `http://www.scilab.org`.

[10] OpenMP Architecture Review Board. OpenMP Application Program Interface v3.1; 2011. Available from: `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`.

[11] NVIDIA. Compute Unified Device Architecture v5.5; 2013. `https://developer.nvidia.com/about-cuda`.

[12] Bezanson J, Karpinski S, Shah VB, Edelman A. Julia: A fast dynamic language for technical computing. arXiv preprint arXiv:12095145. 2012;p. 1–27.

[13] Bradski G. The OpenCV Library; 2000. Dr. Dobb's Journal of Software Tools.

[14] Guennebaud G, Jacob B, et al.. Eigen v3; 2010. `http://eigen.tuxfamily.org`.

[15] Chandra R. Parallel programming in OpenMP. Morgan Kaufmann; 2001.

[16] Lindholm E, Nickolls J, Oberman S, Montrym J. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro. 2008;28(2):39–55.

[17] Harris M, Sengupta S, Owens JD. Parallel prefix sum (scan) with CUDA. GPU gems. 2007;3(39):851–876.

[18] Batcher KE. Sorting networks and their applications. In: 1968 Spring Joint Computer Conference. ACM; 1968. p. 307–314.

[19] Satish N, Harris M, Garland M. Designing efficient sorting algorithms for manycore GPUs. In: Proc. International Symposium on Parallel & Distributed Processing. IEEE; 2009. p. 1–10.

[20] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization. IEEE; 2004. p. 75–86.

[21] Taylor J. Introduction to error analysis, the study of uncertainties in physical measurements. vol. 1. 2nd ed.; 1997.

[22] Mashey JR. War of the benchmark means: time for a truce. ACM SIGARCH Computer Architecture News. 2004;32(4).

[23] Ciemiewicz DM. What Do You 'Mean'? Revisiting Statistics for Web Response Time Measurements. In: Proc. Computer Measurement Group Conference; 2001. p. 385–396.

[24] Lawson CL, Hanson RJ, Kincaid DR, Krogh FT. Basic linear algebra subprograms for Fortran usage. ACM Transactions on Mathematical Software. 1979;5(3):308–323.

[25] Xianyi Z, Qian W, Chothia Z. OpenBLAS; 2012. `http://xianyi.github.io/OpenBLAS`.

[26] The MathWorks, Inc . Accelerating MATLAB; 2002.

[27] Google. Google Performance Tools; 2014. `https://code.google.com/p/gperftools/`.

[28] Nethercote N. Dynamic binary analysis and instrumentation [PhD dissertation]. University of Cambridge; 2004.

[29] de Melo AC. The new Linux `perf` tools. Presented at the Linux Kongress; 2010. .

[30] Hennessy JL, Patterson DA. Computer architecture: a quantitative approach. Elsevier; 2012.