A Modern C++ Point of *View* of Programming in Image Processing (Short paper)

Anonymous Author(s)

Abstract

C++ is multi-paradigm language that enables the programmer to set-up efficient image processing algorithms easily. This language strength comes from many aspects. C++ is high-level so this enables developing powerful abstractions and mixing different programming styles to ease the development. At the same time C++ is low-level and can fully take advantage of the hardware to deliver the best performance. It is also very portable and highly compatible which allows algorithms to be called from high-level, fast-prototyping languages such as Python or Matlab. One of the most fundamental aspects where C++ really shines is *generic programming*. Generic programming makes it possible to develop and reuse bricks of software on objects (images) of different natures (types) without performance loss. Nevertheless, conciliating genericity, efficiency and simplicity at the same time is not trivial. Modern C++ (post-2011) has brought new features that made it simpler and more powerful. In this paper, we will focus in particular on some C++20 aspects of generic programming: ranges, views and concepts and see how they extend to images to ease the development of generic image algorithms while lowering the computation time.

CCS Concepts: • **Software and its engineering** → **Software development techniques**; • **Computing methodologies** → **Image processing**.

Keywords: Image processing, Generic Programming, Modern C++, Software, Performance

ACM Reference Format:
Anonymous Author(s). 2020. A Modern C++ Point of *View* of Programming in Image Processing (Short paper). In *GPCE 2020 - 19th International Conference on Generative Programming: Concepts & Experiences, Nov 15–20, 2020, Chicago, Illinois, US*. ACM, New York, NY, USA, 7 pages. https://doi.org/{FIXME10.1145/1122445.1122456}

GPCE '20, Nov 15–20, 2018, Chicago, Illinois, US

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/{FIXME10.1145/1122445.1122456}

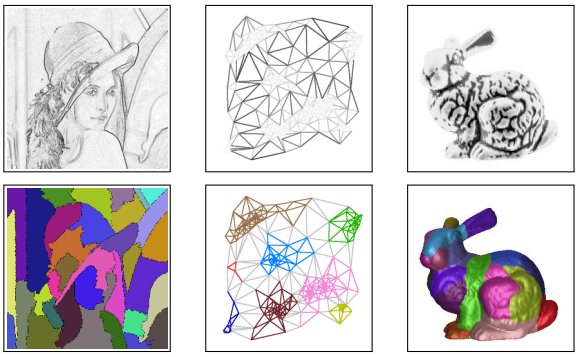

Figure 1. The watershed segmentation algorithm runs on a 2D-regular grayscale image (left), on a vertex-valued graph (middle) and on a 3D mesh (right).

1 Introduction

C++ is claimed to *"leave no room for a lower-level language (except assembler)" [27]* that makes it a go-to language when developing high-performance computing (HPC) image processing applications. The language is designed after a zero-overhead abstraction principle that allows us to devise a high-level but efficient solution to image processing problems. Others aspects of C++ are its stability, its portability on a wide range of architectures, and its direct interface with the C language which makes C++ easily interoperable with high-level prototyping languages. This is why the performance-sensitive features of many image processing libraries (and numerical libraries in general) are implemented in C++ (or C/Fortran as in OpenCV [7], IPP [10]) or with in an hardware-dedicated language (*e.g.* CUDA [8]) and are exposed through an high-level API to Python, LUA…

Apart from the performance considerations, the problem lies in that each image processing field comes with its own set of image type to process. Obviously, the most common image type is an image of RGB or gray-level values, encoded on 8-bits per channel, on a regular 2D rectangular domain that covers 90% of common usages. However, with the development of new devices has come new image types: 3D multi-band images in Medical Imaging, hyperspectral images in Astronomical Imaging, images with complex values in Signal Processing…Some devices generate images with a *depth* channel which is encoded with a number of bits different from the other channels... an image processing library able to handle to images would cover 99% of use cases. Finally, it remains 1% of usages with exoteric image types.

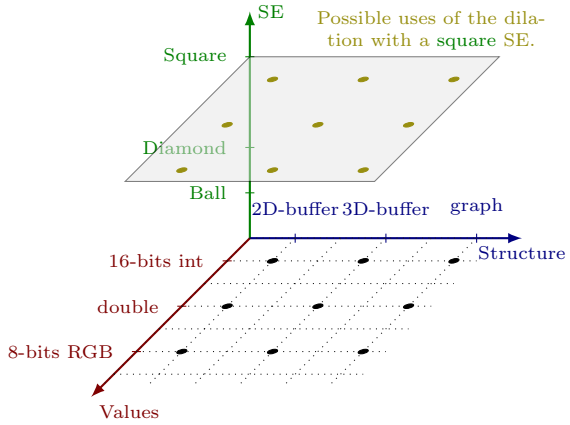

Figure 2. The combinatorial set of inputs that a *dilation* operator may handle.

```
void dilate_rect(image2d_u8 in, image2d_u8 out, int w, int h) {
  for (int y = 0; y < out.height(); ++y)
    for (int x = 0; x < out.width(); ++x) {
      uint8_t s = 0;
      for (int qy = y - h/2; qy <= y + h/2; ++y)
        for (int qx = x - w/2; qx <= x + w/2; ++x)
          if (0 <= qy <= in.height() && 0 <= qx <= in.width())
            s = max(s, input(qx, qy))
      out(x,y) = s;
  }
}
```

Figure 3. Non-generic dilation algorithm for 8-bit grayscale 2D-images by a rectangle

In Digital Topology, we have to deal with non-regular domain, where pixels are *not* regular pixels. They might be super-pixels produced by a segmentation algorithms, might be hexagonal, might be defined on some special grids (*e.g.* the cairo pattern [15]) or meshes. In Mathematical Morphology, most image operators are defined on a graph framework and are naturally extended to hierarchical representations of the image (*e.g.* operators on hierarchies of segmentation [21], on trees [11] or in a shape space [33]). The fact that image processing is related to so many different fields has led to wonder about easily adapting types to fit different image formalisms [17].

From a programming standpoint, the ability of to run the same algorithm (code) over a different set of image types, as shown in fig. 1, is called *polymorphism*. To illustrate our point, we will consider a simple yet complex enough image operation: the *dilation* of an image f by a flat structuring element (SE) $\mathcal{B}$ defined as

$$g(x) = \bigvee_{y \in \mathcal{B}_x} f(y) \tag{1}$$

Simply said, it consists in taking the supremum of the values in region $\mathcal{B}$ centered in x. Despite the apparent simplicity, this operator allows a high variability of the inputs. f can a regular 2D image as well as a graph; values can be grayscale as well as colors; the SE can be rectangle as well as a disc adaptative to the local content... The straightforward implementation in fig. 3 only covers only one possible set of parameters: the dilation of 8-bit grayscale 2D-images by a rectangle. The combinatorial set of parameters increases drastically with the types of the inputs as seen in fig. 2. In [4], the authors depict 4 different approaches to leverage "genericity".

With *Ad-hoc polymorphism (A)*, one has to write one implementation for each image type which involves code duplication to be exhaustive. The ability to select which implementation will run is based on the "real" type of the image. In C++, if this information is known at compile time (*static*), the compiler selects the right implementation by itself (*static dispatch* by *overload resolution*). If the "real" type of the image is known dynamically, one has to select the right by implementation by hand (boilerplate code).

With *Generalization (B)*, one has to consider a common type for all images (let us name it *G*) and write algorithms for this common type. It implies conversion back and forth between *G* and other image types for every computation.

With *Inclusion Polymorphism, Dynamic Traits (C)*, one has to define an abstract type featuring all common image operations. For example, one may consider that all images must define an operator `get_value(Point p) -> Any` where *Point* is a type able to contain any *point value* (2d, 3d, graph vertex...) and *Any* a type able to hold any value. This is generally achieved using *inclusion polymorphism* in Object Oriented Programming with a super-type `AbstractImage` for all image types. It may also be achieved using more modern techniques such as *type-erasure* with a type `AnyImage` (that has the same interface as `AbsractImage`) for which any image could be converted to. Whatever technique used behind the scene relies on a dynamic dispatch at run-time to resolve which interface method is called.

Parametric Polymorphism, Generics, Static Traits (D) somewhat relates to the same concept of *Inclusion Polymorphism*; one also has to define an abstraction for the handled images. However, the main difference lies in the dispatch which is *static* and has the best performance. The compiler writes a new specialized version for each input image type by itself thanks to *template* algorithm. C++ generic programming will be reviewed more in-depth in section 2.

Most libraries do not fall into a single category but mix different techniques. For instance, CImg [28] mixes *(B)* and *(D)* by considering only 4D-images parameterized by their value type. In OpenCV [7], algorithms take *generalized* input types *(C)* but dispatch dynamically and manually on the value type *(A)* to get a *concrete* type and call a generic algorithm *(D)*. Scikit-image [29] relies on Scipy [18] that has a C-style object dynamic abstraction of nd-arrays and iterators *(C)* and sometimes dispatch by hand to the most specialized algorithm based on the element type *(A)*. Many

```
template <Range R>
auto maxof(R col)
requires MaxMonoid<value_t<R>> {
  value_t<R> s = 0;
  for (auto e : col)
    s = max(s, e);
  return s;
}
```

```
template <typename T>
concept MaxMonoid =
  requires(T x) {
  { T v = 0; };
  { x = max(x,x); };
}
```

Figure 4. A generic concept-checked sum algorithm over a collection.

libraries have chosen the *(D)* option with a certain level of genericity (Boost GIL [6], Vigra [20], GrAL [5], DGTal [12], Higra [24], and ANONYM [2]).

The recent advances in the C++ language [26] have eased the development of high-performance code and scientific libraries have taken advantages of these features [16, 22, 32]. The modern C++ has brought *generic programming* to an higher level through *ranges* and *concepts* that enables revisiting what *images* are and what *algorithms* are to develop *generic* algorithms. In particular, it enables mixing *types* and *algorithms* in some new types that are composable. Also, it enables a performance boost while preserving usability that could benefit libraries relying on the (D) approach. The paper is organized as follow. In section 2, we review some basics of *generic programming* and explain how C++20 *concepts* can be used to abstract *image types* in a generic framework. In section 3, we show how the *ranges* proposal and *range views* can be extended to *images* and how they enable the user to gain expressivity and performance. Eventually, in section 4, we validate the performance gain on a real-case benchmark.

2 Algebraic Properties of Images and Related Notions

2.1 The Abstract Nature of Algorithms

Most algorithms are *generic* by nature as demonstrated in the Standard Template Library (STL) [13] when one has to work on a *collection of data*. For example, let us consider the algorithm *maxof(Collection c)* that gets the maximal element of a collection (see fig. 4). It does not matter wether the *collection* is actually implemented with a linked-list, a contiguous buffer of elements or whatever data structure. The only requirements of this algorithm are: (1) we can *iterate* through it; (2) the type of the elements is "regular" and form a monoïd with an associative operator "max" and a neutral element "0". Actually (1) is abstracted by pairs of *iterators* in the STL and *ranges* in C++20, while C++20 introduces *concepts* to check if a type follows the requirements of the algorithm.

2.2 Image Concept

Most image processing algorithms are also *generic* [3, 25] by nature. We saw in section 2.1 that algorithms are defining *concepts* and not the contrary. Similarly to fig. 4, let us

```
template <class I, class SE>
void dilation(I in, I out, SE se) {
  for (auto p : out.domain()) {
    value_t<I> s = min_of_v<value_t<I>>;
    for (auto q : se(p))
       s = max(s, input(q))
    output(p) = s;
  }
}
```

Figure 5. Generic dilation algorithm.

```
template <class I>
concept Image = requires {
  point_t<I>; // Type of point (P)
  value_t<I>; // Type of value (V)
} && requires (I f, point_t<I> p, value_t<I> v) {
  { v = f(p) }; //
  { f(p) = v }; // optional, for output
  { f.domain() } -> Range; // (actually Range of P)
};
template <class SE, class P>
concept StructuringElement =
requires (SE se, P p) {
  { se(p) } -> Range; // (actually Range of P)
};

template <Image I, class SE>
void dilation(I input, I output, SE se)
requires MaxMonoid<value_t<I>>
  && StructuringElement<SE, point_t<I>>
{ ... }
```

Figure 6. Image and Structuring Element concept and constrained version of the dilation algorithm.

consider the morphological dilation of an image $f : E \to F$ (defined on a domain E with values in F) by a flat structuring element (SE) B (we note B_x the SE centered in x). The dilation is defined as $\delta_f(x) = \sup\{f(y),\ y \in B_x\}$; the algorithm is given in fig. 5. As one can see, the implementation does not rely on a specific implementation of images. It could be 2D images, 3D images or even a graph (the SE could be the adjacency relation graph).

The image requirements can be extracted from this algorithm. The image must provide a way to access its domain E which must be iterable. The structuring element must act as a function that returns a range of elements having the same type as the domain element (let us call them *points* of type P). Image has to provide a way to access the value at a given point (`f(x)`) with `x` of type `P`. Last, as in fig. 4, image values (of type V) have to support `max` and have a neutral element "0". It follows the -simplified- *Image* concept and the constrained dilation algorithm in fig. 6. Actually, the requirements for being an image are quite light. This provides versatility and allows us to pass non-regular "image" objects as inputs such as the *image views* in section 3.

2.3 Genericity, Specialization and Performance

It is often argued against generic programming that a single implementation cannot be performance optimal for every types. For example, the generic implementation of the dilation for n-dimensional buffer images convert points into indices to access the data in the buffer while it could use indices directly if the data are contiguous in memory. We claim that this is not the problem of the generic programming paradigm as there exist several algorithms for the same image operator. Performance is the matter of an optimization process, *i.e.* , transforming or adapting the code into an equivalent code that performs better. Some optimizations are within the grasp of compilers mostly low-level ones, while some high-level optimizations are just not reachable by compilers. The dilation operation allows some drastic optimization based on the type of inputs; if the SE is decomposable, use a sequence of dilations with simpler SEs; if the SE is a line, use a dedicated $O(n)$ 1D-algorithm [14], if the data is a contiguous buffer of basic types and the SE is a line, use the 1D vertical dilation with vector processing; if the extent of the SE is small, perform the dilation with a fixed-size mask. C++ GP does not mean that a single implementation will cover all these cases. It cannot as some of these decisions depend on runtime conditions. However, it aims at providing n algorithms to cover m combinations of inputs with $n \ll m$ and ease the selection of the best implementation based on compile-time features of the inputs. Modern-C++ has greatly eased the compile-time selection with concepts and type properties as shown in fig. 7 mixing overload selection with *concept refinement* and *specialization ordering*. Even if the third implementation is very specific to some inputs, it is still generic enough to cover all the native basic types (`float`,`uint8`,`uint16`...) so that we do not have to duplicate code for each of them.

3 Another *View* of Images for Genericity and Performance

C++20 ranges [23] formalizes the concept of *view*, extending the *array views* implemented in array-manipulation libraries[1, 31], and transferable to the *Image* concept. An *image view* is a lightweight object that acts like an image, *i.e.* , it models an *Image*. For example, it can be a random generator image object which generates a value whenever $f(p)$ is called, or an image that records the number of times each pixel is accessed in order to compare algorithms performance... In some pre C++-11 libraries (*e.g.* the GIL [6] or *ANONYM.* [3]), image views were also present in but not compatible with modern C++ idioms (*e.g.* the range-based *for* loop) and not as well developed as in [23] however the idea remain the same and modern C++ ease their development.

Among image views, we give a particular focus on image *adaptors*. Let `v = transform`$(u_1, u_2, \cdots, u_n, h)$ where u_i are input images and h a n-ary function. `transform` returns an

```
template <Image I, class SE> // (1)
void dilation(I input, I output, SE se)
requires MaxMonoid<value_t<I>> &&
         StructuringElement<SE, point_t<I>>
{ /* Generic impl. */ }

template <Image I, class SE>  // (2)
void dilation(I input, I output, SE se)
requires MaxMonoid<value_t<I>> &&
         DecomposableStructuringElement<SE, point_t<I>>
{ /* Decomposition-based impl. */ }

template <class V>  // (3)
void dilation(buffer2d<V> in, buffer2d<V> out, vline2d se)
requires is_arithmetic_v<V>
{ /* SIMD impl. of 1D version */  }
```

Figure 7. Dilation implementation specialization based on compile-time predicates. (1) is the generic fall-back overload, (2) is selected based on constraints ordering and concept refinement of the structuring element; (3) is selected based on the ordering rules for template specializations.

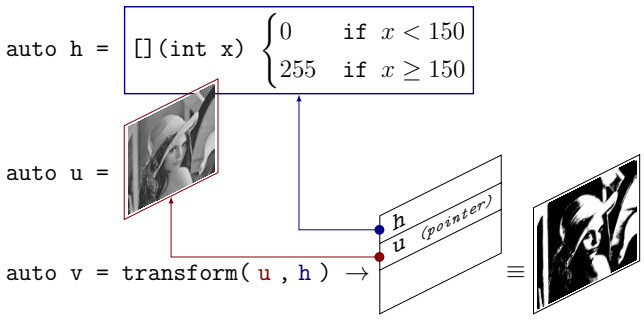

Figure 8. An image view performing a thresholding

image generated (adapting) from other image(s) as shown in fig. 8. An adaptor does not "own" data but records the transformation h and pointers to the input images. The properties of the resulting view depend on h. On one hand, the projection h: $(r, g, b) \mapsto g$ that selects the green component of a RGB triplet gives a view v that is *writable*, with 8-bits integer values and has the same domain as u_1. On the other hand, the projection h: $(a, b) \mapsto (a+b)/2$, applied on images u_1 and u_2 gives a read-only view that computes pixel-wise the average of u_1 and u_2.

Following the same principle, a view can apply a restriction on an image domain. In fig. 9, we show the adaptor `clip(input, roi)` that restricts the image to a non-regular `roi` and `filter(input, predicate)` that restricts the domain based on a predicate. All subsequent operations on those images will only affect the selected pixels.

Views feature many interesting properties that change the way we program an image processing application. To illustrate those features, let us consider the following image processing pipeline: (Start) Load an input RGB-16 2D image (a classical HDR photography) (A) Convert it in grayscale (B)

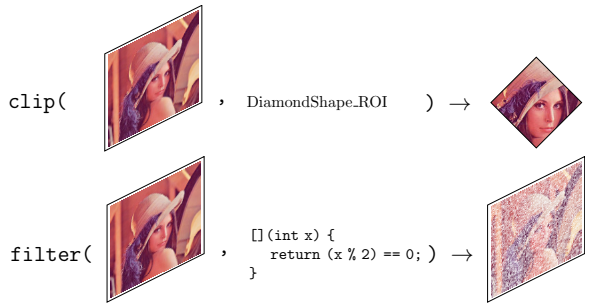

Figure 9. Clip and filter image adaptors that restrict the image domain by a non-regular ROI and by a predicate that selects only even pixels.

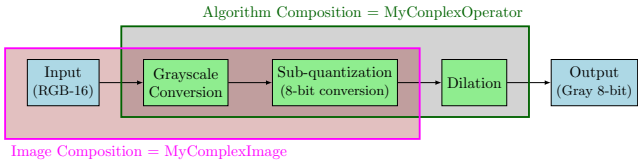

Figure 10. Example of a simple image processing pipeline illustrating the difference between the composition of algorithms and image views.

Sub-quantize to 8-bits (C) Perform the grayscale dilation of the image (End) Save the resulting 2D 8-bit grayscale image; as described in fig. 10.

Views are composable. One of the most important feature in a pipeline design (generally, in software engineering) is *object composition*. It enables composing simple blocks into complex ones. Those complex blocks can then be managed as if they were still simple blocks. In fig. 10, we have 3 simple image operators *Image* → *Image* (the grayscale conversion, the sub-quantization, the dilation). As shown in fig. 10, algorithm composition would consider these 3 simple operators as a single complex operator *Image* → *Image* that could then be used in another even more complex processing pipeline. Just like algorithms, image views are composable, *e.g.* a view of the view of an image is still an image. In fig. 10, we compose the input image with a grayscale transform view and a sub-quantization view that then feeds the dilation algorithm.

Views improve usability. The code to compose images in fig. 10 is almost as simple as:

```
auto input = imread(...);
auto A = transform(input, [](rgb16 x) -> float {
    return (x.r + x.g + x.b) / 3.f; };
auto MyComplexImage = transform(A, [](float x)
    -> uint8_t { return (x / 256 + .5f); };
```

People familiar with functional programming may notice similarities with these languages where *transform* (*map*) and *filter* are sequence operators. Views use the functional paradigm and are created by functions that take a function as argument: the operator or the predicate to apply for each pixel; we do not iterate by hand on the image pixels.

```
auto operator+(Image A, Image B) {
  return transform(A, B, std::plus<>());
}
auto togray = [](Image A) { return transform(A, [](auto x)
  { return (x.r + x.g + x.b) / 3.f; };)
};
auto subquantize16to8b = [](Image A) { return transform(A,
  [](float x) { return uint8_t(x / 256 +.5f); });
};

auto input = imread(...);
auto MyComplexImage = subquantize16to8b(togray(A));
```

Figure 11. Using high-order primitive views to create custom view operators.

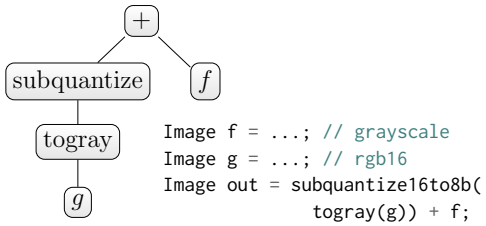

Figure 12. View composition seen as an expression tree.

Views improve re-usability. The code snippets above are simple but not very re-usable. However, following the functional programming paradigm, it is quite easy to define new views, because some image adaptors can be considered as *high-order functions* for which we can bind some parameters. In fig. 11, we show how the primitive *transform* can be used to create a view summing two images and a view operators performing the grayscale conversion as well as the sub-quantization which can be reused afterward[1].

Views for lazy computing. Because the operation is recorded within the image view, this new image type allows fundamental image types to be mixed with algorithms. In fig. 11, the creation of views does not involve any computation in itself but rather delays the computation until the expression `v(p)` is invoked. Because views can be composed, the evaluation can be delayed quite far. Image adaptors are *template expressions* [30, 31] as they record the *expression* used to generate the image as a template parameter. A view actually represents an expression tree (fig. 12).

Views for performance. With a classical design, each operation of the pipeline is implemented on "its own". Each operation requires that memory be allocated for the output image and also, each operation requires that the image be traversed. This design is simple, flexible, composable, but is not memory efficient nor computation efficient. With the lazy evaluation approach, the image is traversed only once (when the dilation is applied) that has two benefits. First, there are no intermediate images so this is memory effective.

[1]Theses functions could have be written in a more generic way for more re-usability but this is not the purpose here

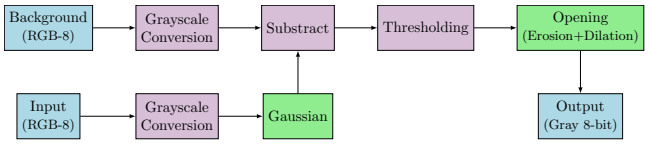

Figure 13. Pipeline for foreground extraction.

Framework	Compute Time	Memory usage	ΔMemory usage
No views (ref)	2.11s *(± 144ms)*	106 MB	***+0%***
OpenCV	2.41s *(± 134ms)*	59 MB	***-44%***
Views	2.13s *(± 164ms)*	51 MB	***-52%***

Table 1. Benchmarks of the pipeline fig. 13 on a dataset (12 images) of 10MPix images. Average computation time and memory usage of implementations with/without *views* and with OpenCV as a baseline.

Second, it is faster thanks to a better memory cache usage; Processing a RGB16 pixel from the dilation algorithm directly convert it in grayscale, then sub-quantized it to 8-bits, and finally make it available. It acts *as if* we were writing an optimal operator that would combine these operations. This approach is somewhat related to the kernel-fusing operations available in some HPC specifications [19] but views-fusion is optimized by the C++ compiler only [9].

As an experiment, we benchmarked both pipelines on a 20MPix image RGB16 (random generated values) on a desktop computer i7-2600 CPU @ 3.40GHz, single-thread[2]. The dilation uses a a small 3x3 square SE with tiling for caching input values. The pipeline using views is about 20% faster than the regular one (133 vs 106 ms). Note that views are also compatibles with other forms of optimizations such as parallelization and vectorization.

4 Experimentation

To hilight the interest of GP and views in the context of performance-sensitive applications, we study the impact on a simple but real case image processing pipeline aiming at extracting objects from a background as depicted on fig. 13. Simply said, it computes the difference between an image and a registered image. The gaussian blur and the morphological opening allow some robustness to noise. The pipeline is implemented with (1) OpenCV, (2) our library where each step is a computing operator, (3) our library where the purple blocks are views. In table 1, we benchmark the compute time and the memory usage [3] of these implementations (all single-threaded) with an opening of disc of radius 32 on 10 MPix RGB images (the minimum of many runs is kept).

[2]Experimentation code is available at *ANONYM*. URL

[3]Memory usage is computed with *valgring/massif* as the difference between the memory peak of the run and the memory peak without any computation (just setup and image loading)

```
float kThreshold = 150; float kVSigma = 10;
float kHSigma = 10;  int kOpeningRadius = 32;
auto img_grey = view::transform(img_color, to_gray);
auto bg_grey  = view::transform(bg_color, to_gray);
auto bg_blurred = gaussian2d(bg_grey,  kHSigma, kVSigma);
auto tmp_grey = img_grey - bg_blurred;
auto thresholdf = [](auto x) { return x < kThreshold; };
auto tmp_bin = view::transform(tmp_grey, thresholdf);
auto ero = erosion(tmp_bin, disc(kOpeningRadius));
dilation(ero, disc(kOpeningRadius), output);
```

Figure 14. Pipeline implementation with views. The lines in blues are the part of the code using *views* obtained by prefixing operators with the namespace `view`.

The results should not be misunderstood. They do not say that OpenCV is faster or slower but shows that implementations all have the same order of processing time (the algorithms used in our implementation are not the same as those used in OpenCV for blur and dilation/erosion) so that the comparison makes sense. It enables to validate experimentally the advantages of views in pipelines. First, we have to be cautious about the real benefit in term of processing time. Here, most of the time is spent in algorithms that are not eligible for view transformation. Thus, depending on the operations of the pipeline, views may not improve processing time. Nevertheless, using views do not degrade performance neither (only 1% in this experiment). It seems to show that using views do not introduce performance penalties and may even be beneficial in lightweight pipelines as the one in section 3. On the memory side, views reduce drastically the memory usage which is beneficial when developing applications with low memory constraints. From the developer standpoint, it requires only few changes in the code as shown in fig. 14 - the implementation of the algorithms remain the same - which is a real advantage for software maintenance.

5 Conclusion

Through simple but concrete examples, we have suggested how modern C++ and the generic programming paradigm can ease image processing software development. We gave a particular focus to the concepts of *image views* and have shown that they improve both performance and usability of an image processing framework. These ideas have been implemented in our C++20 library [2] and used for concrete image processing applications (medical imaging and document analysis). Nonetheless, generic programming in C++ comes with some downsides. Templates belong to the static world and selecting algorithmic specialization based on runtime conditions is not trivial. It requires ahead-of-time generation of specializations that increases compile times and does not scale with the parameter space size, or it requires to switch to a more dynamic paradigm that could degrade performances. Dealing with dynamic conditions is not an option when it comes to expose a static library to a dynamic language like Python. As a future work, we will address this issue.

References

[1] B. Andres, U. Koethe, T. Kroeger, and F.A. Hamprecht. 2010. *Runtime-Flexible Multi-Dimensional Arrays and Views for C++98 and C++0x.* arXiv preprint arXiv:1008.2909. IWR, University of Heidelberg, Germany.

[2] Anonymous author. 2020. ...a Modern C++ Generic Library. https://.

[3] Anonymous authors. 2014. ...Genericity...Algorithms. In *Proc. of a Conference (LNCS).* Springer.

[4] Anonymous authors. 2019. ...Library in Modern C++.... In *Proc. of an Intl. Workshop (LNCS).* Springer.

[5] G. Berti. 2006. GrAL–the Grid Algorithms Library. *Future Generation Computer Systems* 22, 1-2 (2006), 110–122.

[6] L. Bourdev. 2020. Generic Image Library. http://www.lubomir.org/pdfs/GIL_SDJ.pdf.

[7] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* 25 (Nov. 2000), 122–125. https://ci.nii.ac.jp/naid/10028167478/en/

[8] F. Brill and E. Albuz. 2014. NVIDIA VisionWorks toolkit. Presented at the 2014 GPU Technology Conference.

[9] G. Brown, C. Di Bella, M. Haidl, T. Remmelg, R. Reyes, and M. Steuwer. 2018. Introducing Parallelism to the Ranges TS. In *Proceedings of the International Workshop on OpenCL.* 1–5.

[10] I. Burylov, M. Chuvelev, B. Greer, G. Henry, S. Kuznetsov, and B. Sabanin. 2007. Intel Performance Libraries: Multi-Core-Ready Software for Numeric-Intensive Computation. *Intel Technology Journal* 11, 4 (2007).

[11] E. Carlinet and T. Géraud. 2015. MToS: A Tree of Shapes for Multivariate Images. *IEEE Transactions on Image Processing* 24, 12 (2015), 5330–5342.

[12] D. Coeurjolly, J.-O. Lachaud, and B. Kerautret. 2019. DGtal: Digital geometry tools and algorithms library. https://dgtal.org/.

[13] J.C. Dehnert and A. Stepanov. 2000. Fundamentals of Generic Programming. In *Generic Programming (LNCS, Vol. 1766).* Springer, 1–11.

[14] J. Y. Gil and R. Kimmel. 2002. Efficient Dilation, Erosion, Opening, and Closing Algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, 12 (2002), 1606–1617.

[15] B. Grünbaum and G. C. Shephard. 1986. *Tilings and Patterns.* W. H. Freeman & Co.

[16] H. Homann and F. Laenen. 2018. SoAx: A Generic C++ Structure of Arrays for Handling Particles in HPC codes. *Computer Physics Communications* 224 (2018), 325–332.

[17] J. Järvi, M.A. Marcus, and J.N. Smith. 2007. Library Composition and Adaptation Using C++ Concepts. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering.* 73–82.

[18] E. Jones, T. Oliphant, P. Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. http://www.scipy.org.

[19] Khronos Group. 2019. OpenVX. https://www.khronos.org/openvx/.

[20] U. Köthe. 2000. STL-Style Generic Programming with Images. *C++ Report Magazine* 12, 1 (2000), 24–30. https://ukoethe.github.io/vigra.

[21] F. Meyer and J. Stawiaski. 2009. Morphology on graphs and minimum spanning trees. In *Proc. of the Intl. Symp. on Mathematical Morphology (ISMM) (LNCS, Vol. 5720).* Springer, 161–170.

[22] C. Misale, M. Drocco, G. Tremblay, and other. 2018. PiCo: High-Performance Data Analytics Pipelines in Modern C++. *Future Generation Computer Systems* 87 (2018), 392–403.

[23] E. Niebler and C. Carter. 2018. P1037R0: Deep Integration of the Ranges TS. https://wg21.link/p1037r0.

[24] B. Perret, G. Chierchia, J. Cousty, S.J. F. Guimarães, Y. Kenmochi, and L. Najman. 2019. Higra: Hierarchical Graph Analysis. *SoftwareX* 10 (2019), 100335. https://doi.org/10.1016/j.softx.2019.100335

[25] G. X. Ritter, J. N. Wilson, and J. L Davidson. 1990. Image algebra: An overview. *Computer Vision, Graphics, and Image Processing* 49, 3 (1990), 297–331.

[26] R. Smith. 2020. *N4849: Working Draft, Standard for Programming Language C++.* Technical Report. https://wg21.link/n4849.

[27] B. Stroustrup. 2007. Evolving a Language in and for the Real World: C++ 1991-2006. In *Proc. of the 3rd ACM SIGPLAN Conf. on History of Programming Languages* (San Diego, California), Vol. 4. New York, USA, 1–59. https://doi.org/10.1145/1238844.1238848

[28] D. Tschumperlé. 2012. The CImg Library. Online report. https://hal.archives-ouvertes.fr/hal-00927458.

[29] S. van der Walt et al. 2014. Scikit-Image: Image Processing in Python. *PeerJ* (June 2014). https://doi.org/10.7717/peerj.453 DOI 10.7717/peerj.453.

[30] T. L. Veldhuizen. 1995. Expression templates. *C++ Report* 7, 5 (1995), 26–31.

[31] T. L. Veldhuizen. 2000. Blitz++: The Library that Thinks it is a Compiler. In *Advances in Software Tools for Scientific Computing (Lecture Notes on Computational Science and Engineering, Vol. 10).* Springer, 57–87.

[32] M. Werner. 2019. GIS++: Modern C++ for Efficient and Parallel In-Memory Spatial Computing. In *Proc. of the ACM SIGSPATIAL Intl. Workshop on Geospatial Data Access and Processing APIs.* 1–2.

[33] Y. Xu, T. Géraud, and L. Najman. 2015. Connected filtering on tree-based shape-spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38, 6 (2015), 1126–1140.