A Modern C++ Point of View of Programming in Image Processing

Michaël Roynard, Edwin Carlinet, Thierry Géraud

A Modern C++ Point of *View* of Programming in Image Processing

Michaël Roynard
michael.roynard@lrde.epita.fr

Edwin Carlinet
edwin.carlinet@lrde.epita.fr

Thierry Géraud
thierry.geraud@lrde.epita.fr

EPITA Research and Development Laboratory (LRDE)
14-16 rue Voltaire,
94270, Le Kremlin-Bicêtre, France

February 10, 2022

Abstract

C++ is a 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.

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

1 Introduction

C++ claims to *"leave no room for a lower-level language (except assembler)" [38]* which 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 [11]) or with a hardware-dedicated language (*e.g.* CUDA [8]) and are exposed through a high-level API to Python, LUA...

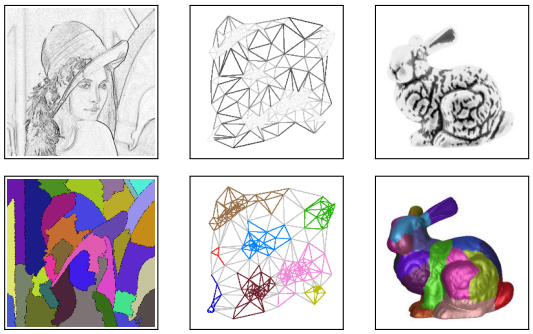

Figure 1: The watershed segmentation algorithm runs on a 2D-regular grayscale image (left), on a vertex-valued graph (middle) and on a 3D mesh (right).

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 with 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 those images type would cover 99% of use cases. Finally, the remaining 1% would cover the usage of esoteric image types.

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 algorithm, hexagonal pixels, pixels defined on some special grids (*e.g.* the cairo pattern [19]) or even meshes' vertices. In Mathematical Morphology, most image operators are defined on a graph

```
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 2: Non-generic dilation algorithm for 8-bits grayscale 2D-images by a rectangle.

framework and are naturally extended to a hierarchical representation of the image (*e.g.* operators on hierarchies of segmentation [28], trees [12] or a shape space [44]). The fact that image processing is related to many fields has already led Järvi to wonder about how they can easily adapt types to fit different image formalism [22].

From a programming standpoint, the ability to run the same algorithm (code) over a different set of image types, as shown in fig. 1, is called *genericity*. This term was defined by Musser in [30] as follows: "*By generic programming we mean the definition of algorithms and data structures at an abstract or generic level, thereby accomplishing many related programming tasks simultaneously. The central notion is that of generic algorithms, which are parametrized procedural schemata that are completely independent of the underlying data representation and are derived from concrete, efficient algorithms.*" 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 be 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 adaptive to the local content... The straightforward implementation in fig. 2 covers only one possible set of parameters: the dilation of 8-bits grayscale 2D-images by a rectangle. The combinatorial set of parameters increases drastically with the types of the inputs as seen in fig. 3. In [34], the authors depict four different approaches to leverage

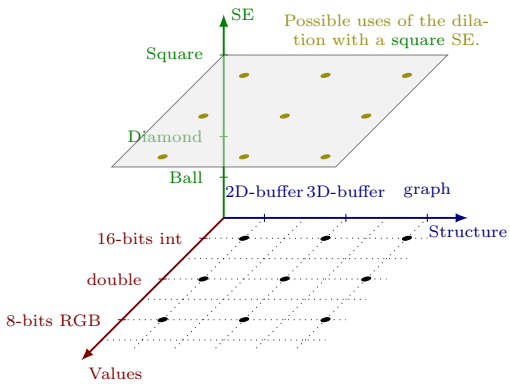

Figure 3: The combinatorial set of inputs that a *dilation* operator may handle.

"genericity" in order to write a generic version of an algorithm.

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 correct implementation by hand by writing boilerplate code.

With *Generalization (B)*, one has to consider a common type for all images (let us name it *super-type*) and write algorithms for this common type. It implies conversion back and forth between the super-type 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 an interface and/or abstract 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 `AbstractImage`) for which any image could be converted to. Whatever technique used behind the

scene relies on a dynamic dispatch at runtime 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 [39] mixes *(B)* and *(D)* by considering only 4D-images parametrized 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 [40] relies on Scipy [23] 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 libraries have chosen the *(D)* option with a certain level of genericity (Boost GIL [6], Vigra [25], GrAL [4], DGTal [14], Higra [32], and Pylene [13]).

The table comparing all the pros and cons from the aforementioned approaches is presented in table 1. We can see in this table that Generic Programming in C++20 check all the boxes that we are interested in.

Table 1: Genericity approaches: pros. & cons.

Paradigm	TC	CS	E	One IA	EA
Code Duplication	✓	✗	✓	✗	✗
Code Generalization	✗	≈	≈	✓	✗
Object-Orientation	≈	✓	✗	✓	✓
Generic Programming:					
with C++11	✓	≈	✓	✓	≈
with C++17	✓	✓	✓	✓	≈
with C++20	✓	✓	✓	✓	✓

TC: type checking; CS: code simplicity; E: efficiency
One IA: one implementation per algorithm; EA: explicit abstractions / constrained genericity

The recent advances in the C++ language [35] have eased the development of high-performance code and scientific libraries have taken advantages of these features [21, 29, 43]. The modern C++ has brought *generic programming* to a higher level through *ranges* [31] and *concepts* [16]. The contributions of this paper are two-fold. First, revisiting the definitions of *images* and *algorithms* to extend *the range views* to *images*. In particular, we enable mixing *types* and *algorithms* in some new types that are composable. Second, we show that it yields performance boost while preserving usability that could benefit libraries relying on the (D) approach.

```
template <Range R>
  requires MaxMonoid<value_t<R>>
auto maxof(R col) {
  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.

The paper is organized as follows. In section 2, we review some basics of *generic programming* and explain how the authors leverages C++20's *concepts* to abstract *image types* by designing a generic framework. In section 3, we present C++20's *ranges*, in particular *range views*, and we contribute by extending this design by applying it to *images*. We also discuss and compare our contribution with state-of-the-art solution that may seem similar to ours in section 4. Eventually, in section 5, 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) [36] 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 whether 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* (*i.e.* behaves the same way as a primitive type like `int`) and forms a monoid with an associative operator

```
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.

"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. The term "concept" is defined as follows in [16]: "*a set of axioms satisfied by a data type and a set of operations on it.*"

While cataloging the image processing operators and algorithms, the authors could extract three main families of algorithms. First are the *pointwise* algorithms which consists in traversing each pixel one by one to perform an operation limited to this pixel (*e.g.* filling the image with a value). Second are the *local* algorithm which consists in traversing each pixel one by one to perform an operation that will consider a window of pixel around this pixel. This window is defined by a *structuring element*. Typical mathematical morphology algorithms such as dilation, closing are part of that family. Finally, are the *global* algorithms which consists in traversing each pixel one by one to perform an operation which may need to consider all the pixels of the image at once, including the previous pixels in the traversing order which have already been transformed. These algorithms are typically propagating a transformation across the whole image. The chamfer distance transformation is a good example of such an algorithm.

When addressing how to write a concept, one should always refer to the following rule: "*It is not the types that define the concepts: it is the algorithms*" [37]. Which means that being able to catalog image processing algorithms mechanically leads to the emergence of concepts related to image processing.

2.2 Image Concept

Most image processing algorithms are also *generic* [33, 26, 27] by nature. We saw in section 2.1 that concepts emerges from pattern behavior extracted from algorithms. Similarly to fig. 4,

```
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.

let us 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 generic 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.

While C++20 provides all the tools necessary to properly define concepts as well as leveraging them when implementing algorithms, it is still necessary to make the inventory of the algorithms families (explained in section 2.1) in order to actually extract the concepts related to image processing. This extracting process is detailed more in-depth

by the authors in [34]. We performed the image processing concept extraction and made it available alongside the image processing library Pylene [13].

2.3 Genericity, Ease of use, Specialization and Performance

It is often argued against generic programming that a single implementation cannot be performance optimal for every type. 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 then use a dedicated $O(n)$ 1D-algorithm [18]; if the data is a contiguous buffer of basic types and the SE is a line then use the 1D vertical dilation with vector processing; if the extent of the SE is small then 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 depends 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 compiletime features of the inputs. Modern C++ has greatly eased the compiletime 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.

Finally, it is often known that there is a rule of three about *genericity*, *performance* and *ease of use*. The rule states that one can only have two

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

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

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

Figure 7: Dilation implementation specialization based on compiletime 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.

of those items by sacrificing the third one. If one wants to be generic and efficient, then the naive solution will be very complex to use with lots of parameters. If one wants a solution to be generic and easy to use, then it will be not very efficient by default. If one wants a solution to be easy to use and efficient then it will not be very generic. This rule can be empirically verified by looking at existing C++ libraries such as [5]. We assert that C++ concepts, used wisely as demonstrated previously enables to break through this rule. A piece of code now can be generic, efficient and easy to use *all at the same time*.

3 Another *View* of Images for Genericity and Performance

3.1 Ranges and Views in C++20 STL

C++20 ranges [31] formalizes the concept of *view*, extending the *array views* implemented in array-manipulation libraries[42, 2], and transferable to the *Image* concept. In the STL, there is a distinction between the container owning the data buffer, the iterators related to traversing this container, the range encapsulating the iterator pair allowing traversing the container and the view which mutates the way the base range traverse the data it is related to. All those abstraction levels need proper

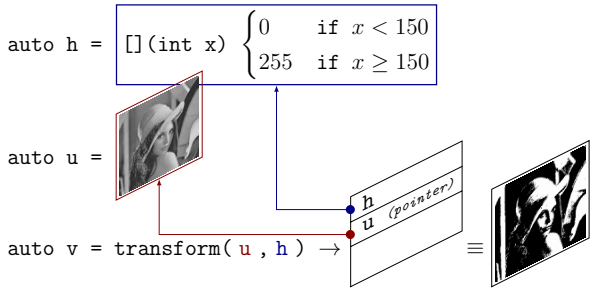

Figure 8: An image view performing a thresholding.

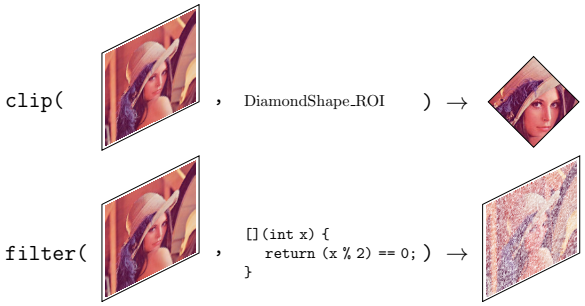

Figure 9: Clip and filter image adaptors that restrict the image domain by a non-regular ROI and by a predicate that selects only even pixels.

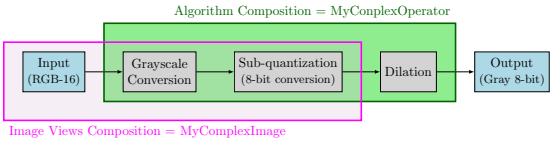

Figure 10: Example of a simple image processing pipeline illustrating the difference between the composition of algorithms and image views.

refined design about data ownership, lifetime of different object depending on what it refers to. For instance, a range may not be cheap-to-copy as it may contain data in order to prolong lifetime of the underlying object, for instance, extending the lifetime of a temporary range in a pipe. Another issue related to range is the semantic of the *constness*. Indeed, the standard has to define what it means for a range view to be *const*. Does it propagate the *constness* to the underlying data or does it impact the view capsule only?

In our design, all images have reference semantics and cheap-to-copy. An *image view*, as a lightweight object that acts like an image, models the *Image* concept. For example, it can be a random generator image object which generates a value whenever $f(p)$ is called, or an *observer* 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 Milena [27]), image views were also present (named *morphers* alongside the SCOOP pattern [10, 17]) but not compatible with modern C++ idioms (*e.g.* the range-based *for* loop) and not as well-developed as in [31] however the idea remains 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 image generated (adapting) from other image(s) as shown in fig. 8. An adaptor does not "own" data but records the transformation h and the pointer to the input images. The properties of the resulting view depend on h. On the one hand, the projection `h`: $(r, g, b) \mapsto g$ that selects the green component of an 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.

3.2 Views applied to image processing

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

Views are composable. One of the most im-

portant 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* $\rightarrow$ *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* $\rightarrow$ *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.

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 operator 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

[1]These functions could have been written in a more generic way for more re-usability, but this is not the purpose here.

```
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.

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* [41, 42] as they record the *expression* used to generate the image as a template parameter. A view actually represents an expression tree (fig. 14).

Views for performance. With a classical design, each operation of the pipeline is implemented on "its own". Each operation requires memory to be allocated for the output image and also, each operation requires that the image is fully 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 which is very memory effective. Second, traversing the image is faster thanks to a better memory cache usage. Indeed, in our example (fig. 10), processing a RGB16 pixel from the dilation algorithm directly converts it in grayscale, then sub-quantize it to 8-bits, and finally makes it available for the dilation algorithm. It acts *as if* we were writing an optimal operator that would combine all these operations. This approach is somewhat related to the kernel-fusing operations available in some HPC specifications [24] but views-fusion is optimized by the C++ compiler only [9].

Views for productivity. All point-wise image processing algorithms can (and should) be rewritten intuitively by using a one-liner view. The *transform* views is the key enabling that point. This implies that there exist a new abstraction level available to the practitioner when prototyping their algorithm. The time spent implementing features is reduced, thus the feedback-loop time is reduced

too. This brings the practitioner to a productivity gain.

3.3 Reasoning at image level

```
void blend_inplace(const uint8_t* ima1, uint8_t* ima2, float alpha,
int width, int height, int stride1, int stride2) {
  for (int y = 0; y < height; ++y) {
    const uint8_t* iptr = ima1 + y * stride1;
    uint8_t* optr = ima2 + y * stride2;
    for (int x = 0; x < width; ++x)
      optr[x] = iptr[x] * alpha + optr[x] * (1-alpha);
  }
}
```

Figure 12: Alpha-blending with classical C/C++ code.

Figure 13: Alpha-blending algorithm written at image level.

The final argument we bring in our discussion about views is the fact that the IP practitioner raises his reasoning by one level. Indeed, let us take a look at the alpha-blending algorithm as a support example for our argument. The default code for a classical, handmade (and error-prone C++) alpha-blending is presented in fig. 12. This algorithm makes several non-relevant hypotheses about the image type. Indeed, it is not relevant to the final application whether the image's color is 8-bits RGB or float. Also, the practitioner may only need to process a specific color channel, or a specific region of the image. The image may also be 3D. To summarize, there are a lot of hypotheses that are not relevant to the application logic and yet weight on the resulting implementations which lead us to the need of genericity. The solution is to shift the abstract level by one layer and reason at image level, as shown in fig. 13 which presents the code and the produced view expression tree. Rewriting the low level algorithm in terms of views is as simple as in fig. 14. Finally, we also show in fig. 15 how simple it now becomes to restrict input images to a specific region or specific color channel directly by chaining views at image level when reasoning at image level. The code has become more *readable*, more *expressive* and more *efficient* by *default*.

```
auto alphablend =
  [](auto ima1, auto ima2, float alpha) {
    return alpha * ima1 +
           (1 - alpha) * ima2;  };
```

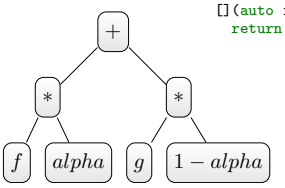

Figure 14: Alpha-blending, generic implementation with views, expression tree.

```
auto ima = blend(ima1, ima2, 0.2); // User-defined view
auto ima_roi = blend(clip(ima1, roi), clip(ima2, roi), 0.2); // ROI
auto ima_red = blend(red(ima1), red(ima2), 0.2); // Red channel
```

Figure 15: Chaining views to feed alpha-blending.

4 Comparison with Data Flow oriented frameworks

A parallel can be drawn between image views and the *data flow* oriented programming [15] style used in Data Science such as the Apache Spark technology [45, 20], Hadoop system [3] or even TensorFlow [1]. Indeed, we find similar properties in those data flow system, such as composition and lazy-computing. Let us focus on the Apache Spark technology for this comparison. This technology is designed in two parts: first is the Spark programming model that creates a dependency graph; second is the runtime system which will schedule work unit on a cluster for the execution of the previously-built graph, and transports code and data to relevant worker nodes.

The spark programming model will proceed in three steps. First is the partitioning function used with a homogeneous collection of objects to construct the Resilient Distributed Dataset (RDD) from our data. Those transformations consist in a pipeline of higher-order functions (*e.g. map*, *filter*, ...), which are chained with each other. Each transformation returns a new RDD which depends on the old RDD. Finally, an action (*reduce*) is performed on the RDD. At that time all the transfor-

mation pipeline is applied on the RDD and computation is scheduled on worker nodes. Furthermore, the Spark programming model allows the developer to fine tune how the program should handle intermediate results (*e.g.* save it on storage for later reuse).

It is very similar to our view design in the fact that transformations can be compared to views (computed lazily and chainable) and actions can be compared to our algorithms (perform the work and resolve the transformation). However, it differs from views at execution time. Views will only do computation on the part of the image that is requested by the final algorithm whereas the data flow pipeline may perform transformation on the whole dataset prior to a narrowing transformation (filtering for instance). The dynamic model enable distribution on clusters of transformations asynchronously when performing actions that acts as barriers in the pipeline, but it does not prevent inefficient and unnecessary computation due to nature of the acyclic computation graph built on the successive transformed RDD. Indeed, RDD are immutable. In contrast, views are static, their composition is static and there is no need of frameworks for that. Also, computation can be done in-place through projector views which is very memory efficient.

Finally, our design differs in the sens that views are *still* image types (with an embedded operation). When reasoning about images, the IP practitioner can focus on behavior of his images and algorithms. On the other hand, Data flow programmer focuses on the data and how to transform it in order to extract information. Design-wise, a RDD is a generalized super-type of data, more flexible due to its dynamic nature, but it does not abstract away the underlying complexity incurred by the processed data.

5 Experimentation

To highlight 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 16. 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 (Pylene) where each step is a computing operator, (3) our library where the purple blocks are views. This pipeline actually produced interesting results, as shown in 17. In table 2, we benchmark the computation time and the memory usage [2]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).

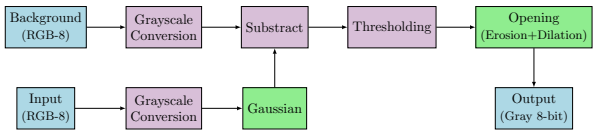

Figure 16: Pipeline for foreground extraction using algorithms and views.

Figure 17: Foreground extraction: sample result.

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

[2]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)

Framework	Compute Time	Memory usage	ΔMemory usage
Pylene (w/o views)	2.11s *(± 144ms)*	106 MB	***+0%***
OpenCV	2.41s *(± 134ms)*	59 MB	***-44%***
Pylene (views)	2.13s *(± 164ms)*	51 MB	***-52%***

Table 2: Benchmarks of the pipeline fig. 16 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.

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

Figure 18: Pipeline implementation with *views*. Highlighted code uses *views* by prefixing operators with the namespace `view`.

OpenCV for blur and dilation/erosion) so that the comparison makes sense. It allows us to validate experimentally the advantages of views in pipelines. First, we have to be cautious about the real benefit in terms 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 does not degrade performance neither (only 1% in this experiment). It seems to show that using views does 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 which are memory constrained. From the developer standpoint, it requires only few changes in the code as shown in fig. 18 — the implementation of the algorithms remain the same — which is a real advantage for software maintenance.

6 Conclusion

Thanks to simple yet concrete examples, we have shown how modern C++ and the generic programming paradigm can ease image processing software development. We have given 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 [13] and used for concrete image processing applications (medical imaging and document analysis). We have compared our design to existing similar design in data flow oriented programming and outlined the main differences. 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 switching to a more dynamic paradigm that could degrade performances. Dealing with dynamic should not be an option when it comes down to exposing a static library to a dynamic language like Python. As a future work, we will research ways to address this issue.

References

[1] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] B. Andres, U. Koethe, T. Kroeger, and F.A. Hamprecht. Runtime-flexible multi-dimensional arrays and views for C++98 and C++0x. arXiv preprint arXiv:1008.2909, IWR, Univ. of Heidelberg, Germany, 2010.

[3] Apache Software Foundation. Hadoop.

[4] G. Berti. GrAL–the grid algorithms library. *Future Generation Computer Systems*, 22(1-2):110–122, 2006.

[5] Boost. Boost c++ libraries.

[6] L. Bourdev. Generic image library. `http://www.lubomir.org/pdfs/GIL_SDJ.pdf`, 2020.

[7] G. Bradski. The OpenCV library. *Dr. Dobb's Journal of Software Tools*, 25:122–125, November 2000.

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

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

[10] Nicolas Burrus et al. A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic

programming. In *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*, Anaheim, CA, USA, October 2003.

[11] I. Burylov, M. Chuvelev, B. Greer, G. Henry, S. Kuznetsov, and B. Sabanin. Intel performance libraries: Multi-core-ready software for numeric-intensive computation. *Intel Technology Journal*, 11(4), 2007.

[12] E. Carlinet et al. MToS: A tree of shapes for multivariate images. *IEEE Transactions on Image Processing*, 24(12):5330–5342, 2015.

[13] E. Carlinet et al. Pylena: a modern C++ image processing generic library, 2018. `https://gitlab.lrde.epita.fr/olena/pylene`.

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

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[16] J.C. Dehnert and A. Stepanov. Fundamentals of generic programming. In *Generic Programming*, volume 1766 of *LNCS*, pages 1–11. Springer, 2000.

[17] Thierry Géraud et al. Semantics-driven genericity: A sequel to the static C++ object-oriented programming paradigm (SCOOP 2). In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, Paphos, Cyprus, July 2008.

[18] J. Y. Gil and R. Kimmel. Efficient dilation, erosion, opening, and closing algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(12):1606–1617, 2002.

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

[20] Dries Harnie et al. Scaling machine learning for target prediction in drug discovery using apache spark. *Future Generation Computer Systems*, 67:409–417, 2017.

[21] H. Homann and F. Laenen. SoAx: A generic C++ structure of arrays for handling particles in HPC codes. *Computer Physics Communications*, 224:325–332, 2018.

[22] J. Järvi, M.A. Marcus, and J.N. Smith. Library composition and adaptation using C++ concepts. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pages 73–82, 2007.

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

[24] Khronos Group. OpenVX. `https://www.khronos.org/openvx/`, 2019.

[25] U. Köthe. STL-style generic programming with images. *C++ Report Magazine*, 12(1):24–30, 2000. `https://ukoethe.github.io/vigra`.

[26] R. Levillain et al. Why and how to design a generic and efficient image processing framework: The case of the Milena library. In *Proceedings of the IEEE Intl. Conf. on Image Processing (ICIP)*, pages 1941–1944, Hong Kong, 2010.

[27] R. Levillain et al. Practical genericity: Writing image processing algorithms both reusable and efficient. In *Proc. of the 19th Iberoamerican Congress on Pattern Recognition (CIARP)*, volume 8827 of *LNCS*, pages 70–79. Springer, 2014.

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

[29] C. Misale, M. Drocco, G. Tremblay, and other. PiCo: High-performance data analytics pipelines in modern C++. *Future Generation Computer Systems*, 87:392–403, 2018.

[30] David R. Musser and Alexander A. Stepanov. Generic programming. In *Intl. Symp. on Symbolic and Algebraic Computation*, pages 13–25. Springer, 1988.

[31] E. Niebler and C. Carter. P1037R0: Deep integration of the ranges TS, May 2018. `https://wg21.link/p1037r0`.

[32] B. Perret, G. Chierchia, J. Cousty, S.J. F. Guimarães, Y. Kenmochi, and L. Najman. Higra: Hierarchical graph analysis. *SoftwareX*, 10:100335, 2019.

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

[34] M. Roynard, E. Carlinet, and T. Géraud. An image processing library in modern C++: Getting simplicity and efficiency with generic programming. In *Reproducible Research in Pattern Recognition—2nd Intl. Workshop*, volume 11455 of *LNCS*, pages 121–137. Springer, 2019.

[35] R. Smith. N4849: Working draft, standard for programming language C++. Technical report, January 2020. `https://wg21.link/n4849`.

[36] Alexander Stepanov and Meng Lee. *The standard template library*, volume 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995.

[37] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, jun 2009.

[38] B. Stroustrup. 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*, volume 4, pages 1–59, New York, USA, 2007.

[39] D. Tschumperlé. The CImg library. Online report, June 2012. `https://hal.archives-ouvertes.fr/hal-00927458`.

[40] S. van der Walt et al. Scikit-Image: Image processing in Python. *PeerJ*, June 2014. DOI 10.7717/peerj.453.

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

[42] T. L. Veldhuizen. Blitz++: The library that thinks it is a compiler. In *Advances in Software Tools for Scientific Computing*, volume 10 of *Lecture Notes on Computational Science and Engineering*, pages 57–87. Springer, 2000.

[43] M. Werner. 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*, pages 1–2, 2019.

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

[45] Matei Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012. USENIX Association.