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

Michaël Roynard
michael.roynard@epita.fr
EPITA Research Laboratory (LRE)
Le Kremlin-Bicêtre, France

Edwin Carlinet
edwin1.carlinet@epita.fr
EPITA Research Laboratory (LRE)
Le Kremlin-Bicêtre, France

Thierry Géraud
thierry.geraud@epita.fr
EPITA Research Laboratory (LRE)
Le Kremlin-Bicêtre, France

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 fundamental aspects where C++ 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 focus 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:
Michaël Roynard, Edwin Carlinet, and Thierry Géraud. 2022. A Modern C++ Point of *View* of Programming in Image Processing. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3564719.3568692

GPCE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9920-3/22/12…$15.00
https://doi.org/10.1145/3564719.3568692

1 Introduction

C++ claims to *"leave no room for a lower-level language (except assembler)" [30]* 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. Other 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. Therefore, many image processing libraries (and numerical libraries in general) implement performance-sensitive features in C++ (or C/Fortran as in OpenCV [4], IPP [8]) or with a hardware-dedicated language (*e.g.* CUDA [5]), which are exposed through a high-level API to Python, LUA, etc.

Apart from the performance considerations, the problem lies in that each image processing field comes with its own set of image type to process. The most common image type is an image of RGB or gray-level values, encoded with 8-bits channel, on a regular 2D rectangular domain. That covers 90% of common usages. However, new image types have come with the development of new devices: 3D multi-band images in Medical Imaging, hyperspectral images in Astronomical Imaging, images with complex values in Signal Processing. An image processing library able to handle those images

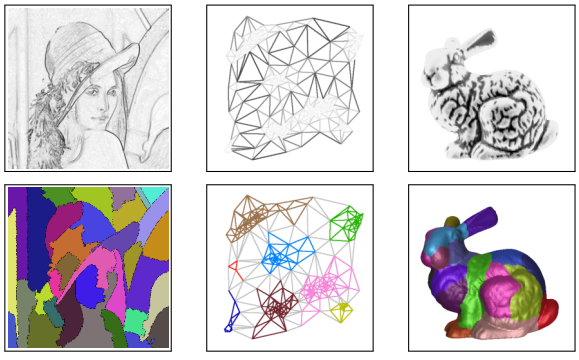

Figure 1. The watershed segmentation algorithm runs on a 2D-regular grayscale image (left), on a vertex-valued graph (middle) and on a 3D mesh (right).

```
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, in(qx, qy));
      out(x,y) = s;
  }
}
```

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

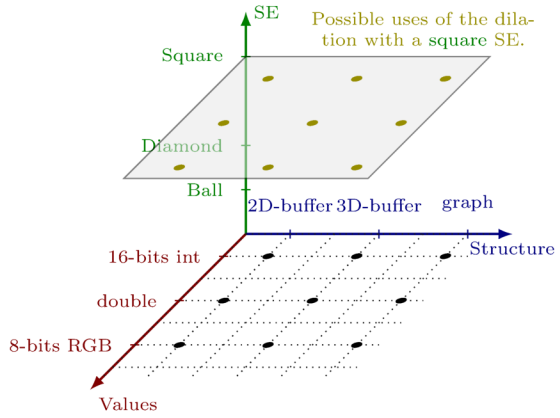

Figure 3. The combinatorial set of inputs that a *dilation* operator may handle.

type would cover almost all use cases. The remaining few is about esoteric image types support.

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 [21] as follows: "*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 with the *least upper bound operator* $\vee$ 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 for high variability of the input data. 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 input data, as seen in fig. 3.

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

In [26], the authors depict four different approaches to leverage "genericity" in order to write a generic version of an algorithm. They are *Ad-hoc polymorphism (A), Generalization (B), Inclusion Polymorphism, Dynamic Traits (C)* and *Parametric Polymorphism, Generics, Static Traits (D).*

Most libraries do not fall into a single category but mix different techniques. For instance, CImg [31] mixes *(B)* and *(D)* by considering only 4D-images parametrized by their value type. In OpenCV [4], 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 [32] relies on Scipy [16] 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 [3], Vigra [18], DGTal [10], Higra [23], and Pylene [9]).

table 1 compares all the pros and cons from the aforementioned approaches. We can see that Generic Programming in C++20 checks all the boxes that we are interested in.

The recent advances in the C++ language [28] have brought *generic programming* to a higher level through *ranges* [22] and *concepts* [13]. This allows us to achieve genericity while reconciling efficiency and ease of use. To our knowledge, there is no other IP libraries that leverage modern C++ features as we do in Pylene to reason at image level instead of reasoning at pixel level, which is closer to the way IP practitioners program in Python. This challenge is relevant in particular for integrators when porting prototype code to production code. We offer a solution to simply write efficient and maintainable code. The contributions of this paper are two-fold. First, in section 3 we revisit the definitions of *images* and *algorithms* to extend *the range views* to *images* in order to achieve reasoning at image level when writing C++ IP algorithms. In particular, we enable mixing *types* and *algorithms* in new composable types. Second, in section 4 we validate the performance gain on a real-case benchmark.

```
template <Range R>
  requires MaxMonoid<value_t<R>>
auto maxof(R col) {
  value_t<R> s = min_of_v<value_t<R>>;
  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. Concept-checked sum algorithm over a collection.

```
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, in(q))
    out(p) = s;
  }
}
```

Figure 5. Generic dilation algorithm.

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) [29] where 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). Whether the *collection* is actually implemented with a linked-list, a contiguous buffer of elements or whatever data structure is irrelevant. The only requirements for 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 "max" and a neutral element "`std::numeric_limits<value_t<R>>::min()`". A monoid is an abstract algebraic structure equipped with an associative binary operation and an identity element. Such structure is studied to achieve high efficiency in several areas, such as distributed computation of small scale algorithms [11] calculus of object query languages [14]. There exists other algebraic structures, such as semi-ring dictionaries [27] that are studied to unify performance optimizations used in both database and linear algebra.

The constraint (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 [13]: "*a set of axioms satisfied by a data type and a set of operations we can perform on it.*". Concepts enable checking the requirements for (2).

2.2 The Image Concept

Most image processing algorithms are also *generic* [19, 20, 25] by nature. We saw in section 2.1 that concepts emerges from pattern behavior extracted from algorithms. Similarly to fig. 4, let us consider the morphological dilation of an image $f : E \longrightarrow F$ (defined on a domain E with values in F) by

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

Figure 6. Image concept and constrained dilation algorithm.

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 the image. It could be a 2D image, a 3D image or even a graph image (the SE would be the adjacency relation of the 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 needs to provide a way to access its value for 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". We deduce 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 [26]. We performed the image processing concept extraction and made it available alongside the image processing library Pylene [9].

3 *Image Views*: Ease and Performance

3.1 Ranges and Views in the C++20 STL

C++20 ranges [22] formalizes the concept of *view*, extending the *array views* implemented in array-manipulation libraries [2, 35], 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 refined design about data ownership and

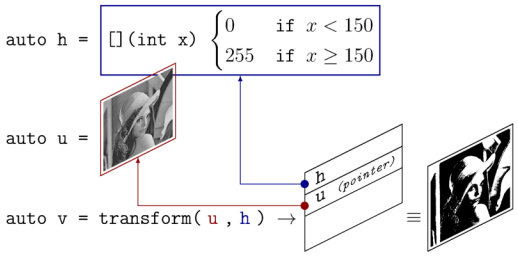

Figure 7. An image view performing a thresholding.

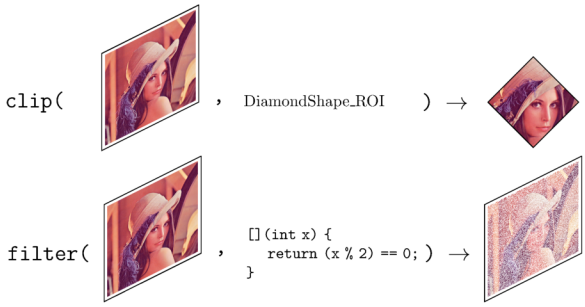

Figure 8. Clip and filter image adaptors that restrict the image domain by a non-regular ROI and by a predicate that selects only even pixels.

lifetime, depending on what it refers to. For instance, a range may not be cheap-to-copy as it may contain data to prolong lifetime of the underlying object, as for extending the lifetime of a temporary range in a pipe.

In our design, all images have reference semantics and are cheap-to-copy. An *image view*, is a lightweight object that acts like an image and models the *Image* concept. For example, it can be a generator image object which generates a new value whenever $f(p)$ is called, or an *observer* image that records the number of times each pixel is accessed to compare algorithms performance. In some pre C++-11 libraries (*e.g.* GIL [3] or Milena [20]), image views were also present (named *morphers* alongside the SCOOP pattern [7, 15]) but not compatible with modern C++ idioms (*e.g.* the range-based *for* loop) and not as well-developed as in [22].

Among image views, we focus on image *adaptors*. Let `v = transform(`$u_1, u_2, \cdots, u_n, h$`)` where u_i are input images and h an n-ary function. `transform` returns an image generated (adapting) from other image(s) as shown in fig. 7. An adaptor does not "own" data but stores the transformation h and references to the input images. The properties of the resulting view depend on h. The projection `h1`: $(r, g, b) \mapsto g$ that selects the green component of an RGB triplet gives a view that is *writable* and has the same domain as u_1. However, the projection `h2`: $(a, b) \mapsto (a + b)/2$, 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. 8, we show the adaptor

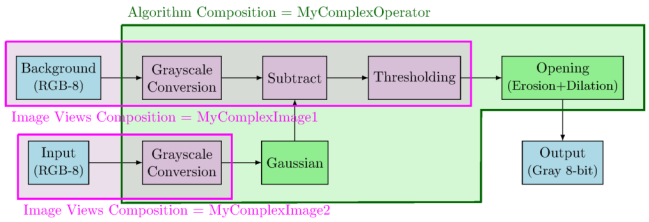

Figure 9. Pipeline for foreground extraction using algorithms and views.

```
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 10. Pipeline implementation with *views*.

`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. Let us consider the simple but real case image processing pipeline aiming at extracting objects from a background which is depicted in fig. 9. Simply said, it computes the difference between an image and a background image. The Gaussian blur and the morphological opening allow some robustness to noise. All the concepts and techniques shown here can be generalized to more complicated IP pipelines.

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. 9, we have 5 simple image operators *Image* → *Image* (grayscale conversion, subtract, thresholding, opening, Gaussian). As shown with *MyComplexOperator*, algorithm composition would consider these 5 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. As shown with *MyComplexImage1 & 2*, we pipe the input images into other views before feeding them into algorithms.

Views improve usability. The code to compose images in fig. 9 is as simple as what is shown in fig. 10. 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 pixel-wise; we do not iterate by hand on the image pixels. Also, it offers the opportunity to raise the IP practitioner reasoning by one level. Indeed, in our example fig. 10, the thresholding routine is written without any consideration of the dimension of the image (2D, 3D), nor the underlying data-type (8-bits, float, RGB). Thanks to this new abstraction level, we are able to express more complex image processing routines very smoothly while remaining generic by default. For instance, restricting the input image to a specific region or to a specific color channel becomes as simple as writing the following code:

```
auto imroi = thresholdf(view::clip(img_gray, roi)); // ROI
auto imred = thresholdf(view::red(img_gray));//Red channel
```

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. 10, the creation of views does not involve any computation in itself but rather delays the computation until the expression $v(p)$ is invoked (requesting the value for the point p). Because views can be composed, the evaluation can be delayed quite far. Image adaptors are *template expressions* [34, 35] as they record the *expression* used to generate the image as a template parameter. A view actually represents an expression tree (cf. fig. 13). This approach is close to designing a Domain Specific Language (DSL) [33], but it remains within the C++ languages, which is unlike another DSL for Image Processing named Halide [24] that have chosen to provide their toolchain infrastructure to solve performance issues related to heterogeneous computing.

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. That has three benefits. First, there are no intermediate images, which is very memory effective. Second, traversing the image maybe be faster thanks to a better memory cache usage. Indeed, a view acts *as if* we were writing an optimal operator that would combine all the operations. This approach is somewhat related to the kernel-fusing operations available in some HPC specifications [17] but views-fusion is directly optimized by the C++ compiler [6]. Also, as we saw that the operations are represented by a functional Abstract Syntax Tree (AST), the compiler is able to perform optimization based on the deforestation algorithm [36] that removes intermediate computation trees in order to achieve a *treeless form*. Third, and the most important, if there is a domain restricting operation downstream in a view pipeline, the operation upstream will only be performed on the relevant region of interest, instead of being performed on the whole image if there was no lazy-computing involved.

```
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 11. Alpha-blending with classical C/C++ code.

Figure 12. Alpha-blending algorithm written at image level.

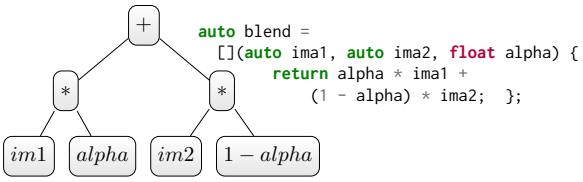

Figure 13. Alpha-blending, generic implementation with views, expression tree.

3.3 Reasoning 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. 11. 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. 12 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. 13. Finally, we also show in fig. 14 how simple it now becomes to restrict input images to a specific region or specific color channel directly by chaining views at image level thanks to being able to reason at image level. The code has become more *readable*, more *expressive* and more *efficient* by *default*.

```
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 14. Chaining views to feed alpha-blending.

Figure 15. Foreground extraction: sample result.

3.4 Comparison with Data Flow Oriented Frameworks

A parallel can be drawn between image views and the *data flow* oriented programming [12] style used in Data Science such as Apache Spark [37] or even TensorFlow [1]. Indeed, we find similar properties in those data flow system, such as composition and lazy-computing. Data flow oriented programming is heavily inspired from functional programming collections (Scala, Haskell). Let us focus on the Apache Spark programming model for this comparison.

It is very similar to our view design in the fact that transformations (partitioning functions) can be compared to views (computed lazily and chainable) and actions (performed on Resilient Distributed Dataset (RDD) constructed from transformations) can be compared to our algorithms (perform the work and resolve the transformation). However, it differs from views at runtime. Views may only do computation on a specific requested part of an image (as seen in section 3.2) whereas the Spark pipeline may do transformations on the whole dataset prior to performing a narrowing transformation. Indeed, Spark aims at distributing asynchronously computation on clusters, that is why it does not prevent inefficient and unnecessary computation due to the nature of the acyclic computation graph computed from RDDs. In contrast, views are static, and their compositions is static. There is no need for a framework for that. Also, Views enables in-place computation (through projectors) which is very memory efficient.

Finally, our design differs in the sens that views are *still* image types (embedding an operation). The IP practitioner can focus on the behavior of his images and algorithms by reasoning at image level. On the other hand, data flow programmer focuses on the data and how to transform it in order to extract information. Design-wise, an 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.

Table 2. Benchmarks of the pipeline fig. 9 on a dataset (12 images) of 10MPix images with OpenCV as a baseline.

Framework	Compute Time (σ)	Memory usage (Δ)	Binary size (Δ)
Ours (no view)	2.11 s ($\pm$ 144 ms)	106 MB (***+0%***)	3.3 MB (***+0%***)
Ours (views)	2.13 s ($\pm$ 164 ms)	51 MB (***-52%***)	2.7 MB (***-17%***)
OpenCV	2.41 s ($\pm$ 134 ms)	59 MB (***-44%***)	2.9 MB (***-11%***)

4 Experimentation

To highlight the interest of Generic Programming 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. 9. 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) and our library where the purple blocks are views. This pipeline actually produces interesting results, as shown in fig. 15. In table 2, we benchmark the computation time and the memory usage (using *valgring/massif*) 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).

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, despite the fact that 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 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. On the binary size side, while we can fear "code bloat" due to many template instanciations, it appears that our binary size is very much in the same order of magnitude as for one doing the same work with OpenCV. We can deduce that the compiler is able to prune generated code and keep only needed and optimized machine code in the final binary. From the developer standpoint, it requires only few changes in the code as shown in fig. 10 — the implementation of the algorithms remains the same — which is a real advantage for software maintenance.

To conclude, the main advantage of views lies in the usability gain for the user without degrading performance. They can even improve memory usage efficiency, which is useful for embedded systems. To showcase this point, we offer additional material [1] demonstrating how views can, for instance, be used for a connected-component labeling algorithm. On a toy example, (a simpler pipeline) we show that views can improve performance (the source is in the supplementary material).

5 Limitations and Conclusion

This paper demonstrates the usage of *image views*, a modern C++ feature applied to image processing, that reconciles genericity, efficiency and ease of use. This contribution allows the IP practitioner to reason at image level, which is novel in the C++ IP library ecosystem, and facilitates the transition between prototypes and production-ready applications. In addition to improving the usability, it leads to performance gain and decreases the memory usage. These ideas have been implemented in our C++20 library [9] and used for concrete image processing applications (medical imaging and document analysis).

Despite these advantages, this design comes with some limitations. The intensive use of the C++ templates generates type combinatorial explosion and code bloat that leads to large compilation times. Also, templates belong to the static world and are poorly interoperable with the dynamic world. Nevertheless, dealing with the dynamic world (runtime) is mandatory when it comes down to exposing a static library to a dynamic language like Python, in order to provide the interactivity that C++ lacks. Another drawback lies in relying on the compiler to do a lot of work, especially the optimizer. Therefore, when the compiler fails in optimizing our abstractions away, we pay the price in performance. Our future work will be dedicated to fix these shortcomings.

References

[1] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[2] B. Andres et al. 2010. *Runtime-Flexible Multi-Dimensional Arrays and Views for C++98 and C++0x.* arXiv preprint. IWR, Univ. of Heidelberg, Germany.

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

[4] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* 25 (12 2000), 122–125. https://ci.nii.ac.jp/naid/10028167478/en/

[5] F. Brill et al. 2014. NVIDIA VisionWorks toolkit. Presented at the 2014 GPU Technology Conference.

[6] G. Brown et al. 2018. Introducing Parallelism to the Ranges TS. In *Proceedings of the International Workshop on OpenCL*. 1–5.

[7] Nicolas Burrus et al. 2003. 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.

[8] I. Burylov et al. 2007. Intel Performance Libraries: Multi-Core-Ready Software for Numeric-Intensive Computation. *Intel Technology Journal* 11, 4 (2007).

[9] E. Carlinet et al. 2018. Pylena: a Modern C++ Library for Generic and Efficient Image Processing. https://gitlab.lrde.epita.fr/olena/pylene

[10] D. Coeurjolly et al. 2019. DGtal: Digital geometry tools and algorithms library. https://dgtal.org/

[11] Ben Dean. 2019. Identifying Monoids: Exploiting Compositional Structure in Code. Slides available at https://github.com/boostcon/cppnow_presentations_2019/blob/master/05-09-2019_thursday/Identifying_Monoids_Exploiting_Compositional_Structure_in_Code__Ben_Deane_cppnow_05092019.pdf, Lecture available on youtube at https://youtu.be/INnattuluiM.

[12] Jeffrey Dean et al. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (01 2008), 107–113. https://doi.org/10.1145/1327452.1327492

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

[14] Leonidas Fegaras and David Maier. 1995. Towards an Effective Calculus for Object Query Languages. *SIGMOD Rec.* 24, 2 (may 1995), 47–58. https://doi.org/10.1145/568271.223789

[15] Thierry Géraud et al. 2008. 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.

[16] E. Jones et al. 2001–. SciPy: Open Source Scientific Tools for Python. http://www.scipy.org

[17] Khronos Group. 2019. OpenVX.

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

[19] R. Levillain et al. 2010. 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)*. Hong Kong, 1941–1944.

[20] R. Levillain et al. 2014. Practical Genericity: Writing Image Processing Algorithms both Reusable and Efficient. In *Proc. of the 19th Iberoamerican Congress on Pattern Recognition (CIARP) (LNCS)*. Springer.

[21] David R. Musser et al. 1988. Generic programming. In *Intl. Symp. on Symbolic and Algebraic Computation*. Springer, 13–25.

[22] E. Niebler et al. 2018. P1037R0: Deep Integration of the Ranges TS. https://wg21.link/p1037r0

[23] B. Perret et al. 2019. Higra: Hierarchical Graph Analysis. *SoftwareX* 10 (2019), 100335. https://doi.org/10.1016/j.softx.2019.100335

[24] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPLAN Notice* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[25] G. X. Ritter et al. 1990. Image Algebra: An Overview. *Computer Vision, Graphics, and Image Processing* 49, 3 (1990), 297–331.

[26] M. Roynard et al. 2019. An Image Processing Library in Modern C++: Getting Simplicity and Efficiency with Generic Programming. In *2nd Intl. Workshop on Reproducible Research in Pattern Recognition (LNCS, Vol. 11455)*. Springer, 121–137.

[27] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–33. https://doi.org/10.1145/3527333

[28] R. Smith. 2020. *N4849: Working Draft, Standard for Programming Language C++*. Technical Report.

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

[1] Supplementary material available at https://www.lrde.epita.fr/~mroynard/gpce.2022/supplementary_material.html

[30] 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

[31] D. Tschumperlé. 2012. The CImg Library. Online report.

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

[33] Arie van Deursen, Paul Klint, and Joost Visser. 2000. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Not.* 35, 6 (06 2000), 26–36. https://doi.org/10.1145/352029.352035

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

[35] 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.

[36] Philip Wadler. 1988. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, H. Ganzinger (Ed.). Springer Berlin Heidelberg, 344–358.

[37] Matei Zaharia et al. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, 15–28.

Received 2022-08-12; accepted 2022-10-10