Une approche générique du logiciel pour le traitement d'images préservant les performances

Roland LEVILLAIN[1,2], Thierry GÉRAUD[1,2]*, Laurent NAJMAN[2]

[1]Laboratoire de Recherche et Développement de l'EPITA (LRDE)
14-16, rue Voltaire, 94276 Le Kremlin-Bicêtre Cedex, France

[2]Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge
Équipe A3SI, ESIEE Paris, Cité Descartes, BP 99, 93162 Noisy-le-Grand Cedex, France
roland.levillain@lrde.epita.fr, thierry.geraud@lrde.epita.fr, l.najman@esiee.fr

Résumé – L'approche générique du logiciel en traitement d'image permet d'écrire des algorithmes réutilisables, compatibles avec de nombreux types d'entrées. Cependant, ce choix de conception se fait souvent au détriment des performances du code exécuté. Du fait de la grande variété des types d'images existants et de la nécessité d'avoir des implémentations rapides, généricité et performance apparaissent comme des qualités essentielles du logiciel en traitement d'images. Cet article présente une approche permettant l'écriture de variantes d'algorithmes basées sur les caractéristiques des types de données utilisés, offrant un compromis entre généricité et performance. Il est possible d'obtenir des temps d'exécutions comparables à ceux d'un code dédié, et dans certains cas de surpasser des routines optimisées manuellement.

Abstract – The generic approach to image processing software allows users to write reusable algorithms compatible with many input types. However, this design choice is often made at the expense of some performance loss. The great variety of image types and the need for fast implementations make both genericity and efficiency desirable features for image processing software. This paper presents an approach enabling the definition of algorithm variants based on data types features, offering an adjustable trade-off between genericity and efficiency. Such variants can match dedicated code in terms of execution times, and may sometimes perform better than routines optimized by hand.

1 Introduction

À l'instar de nombreux domaines de calcul scientifique, le traitement d'images (TDI) doit faire face à deux types de problèmes difficiles à résoudre simultanément. D'une part le spectre des types de données à traiter peut être très large : les méthodes de TDI peuvent être utilisées avec des images à deux dimensions « classiques » composées des pixels carrés organisés selon une grille régulière, contenant des valeurs binaires, en niveaux de gris, en couleurs, vectorielles ou même tensorielles ; la dimension peut également changer : 1D (signaux), 3D (volumes), 2D+t (séquences d'images), 3D+t (séquences de volumes) ; le domaine de l'image peut de plus être non régulier : il est possible de définir des images sur des structures mathématiques telles que des graphes, des complexes simpliciaux ou des cartes topologiques. La capacité à prendre en compte autant de structures de données différentes dépend de la *généricité* du cadre théorique choisi, et s'agissant de l'implémentation, des outils logiciels associés. On désigne par le terme *algorithme générique* un algorithme qui peut être appliqué à des entrées diverses, par opposition à un *algorithme spécifique*, qui ne peut être employé qu'avec une structure de données précise. La programmation générique (PG) est un paradigme de programmation qui s'intéresse aux question de généricité dans le logiciel [1]. L'une des motivations derrière la PG est la *réutilisabilité* du logiciel, c'est-à-dire la minimisation du coût lié à l'utilisation d'algorithmes existants avec de nouvelles structures de données (types d'entrées) et vice versa. Il existe plusieurs projets logiciels dédié au TDI qui reposent sur la PG : Vigra [4], ITK [3], Morph-M [2], GIL [1].

D'autre part, de nombreux problèmes et applications du TDI font intervenir des jeux de données important (par leur nombre ou leur taille) ou font appel à des techniques complexes nécessitant des calculs intensifs. Dans les deux cas, toute solution logicielle doit en pratique satisfaire des contraintes de *performance*, en particulier vis-à-vis de la vitesse d'exécution. Malheureusement, généricité et performance sont souvent opposés : un programme performant de TDI est le plus souvent dédié à certains types d'images, certaines méthodes ou certains domaines d'application spécifiques, et n'est de fait pas générique. À l'inverse, la plupart des *frameworks* génériques ne rivalisent pas avec des solutions dédiées en termes de performances.

Dans cet article, nous examinons les problèmes qui opposent

*Ce travail a été effectué dans le cadre du projet SCRIBO (http://www.scribo.ws/) du Groupe Thématique Logiciel Libre du pôle de compétitivité "Systematic Paris-Région". Ce projet est partiellement financé par le Gouvernement Français, ses agences de développement économique ainsi que les institutions de la région Île-de-France.

1. Dans cet article, par « générique » nous entendons « générique vis-à-vis de la programmation », et non « générique par rapport à l'approche utilisée pour résoudre un problème de traitement d'images ».

```
image dilation(const image& input) {
  image output(input.nrows(), input.ncols());
  for (unsigned r = 0; r < input.nrows(); ++r)
    for (unsigned c = 0; c < input.ncols(); ++c) {
      unsigned char sup = input(r,c);
      if (r != 0 && input(r-1,c) > sup)
        sup = input(r-1,c);
      if (r != input.nrows()-1 && input(r+1,c) > sup)
        sup = input(r+1,c);
      if (c != 0 && input(r,c-1) > sup)
        sup = input(r,c-1);
      if (c != input.ncols()-1 && input(r,c+1) > sup)
        sup = input(r,c+1);
      output(r, c) = sup;
    }
  return output;
}
```

ALGORITHME 1 – Implémentation de dilatation non générique

```
template <typename I, typename W>
I dilation(const I& input, const W& win) {
  I output; initialize(output, input);
  // Itérateur sur les sites du domaine de 'input'.
  mln_piter(I) p(input.domain());
  // Itérateur sur les voisins de 'p' selon 'win'.
  mln_qiter(W) q(win, p);
  for_all(p) {
    // Accumulateur calculant le supremum de 'win'.
    accu::supremum<mln_value(I)> sup;
    for_all(q) if (input.has(q))
      sup.take(input(q));
    output(p) = sup.to_result();
  }
  return output;
}
```

ALGORITHME 2 – Implémentation de dilatation générique [5]

généricité et performance dans le TDI d'un point de vue algorithmique (les optimisations bas niveau ou matérielles ne sont pas abordées). Nous présentons tout d'abord comment la PG peut être appliquée au TDI et quels sont les bénéfices de cette approche (section 2). La section 3 aborde les questions de performance dans un contexte générique. Nous y étudions les raisons de l'opposition entre généricité et performance et nous proposons dans la section 4 une réponse à ce problème sous la forme d'un compromis. Cette idée développée plus encore dans la section 5, où non seulement les algorithmes, mais aussi les types de données, sont mis à contribution pour augmenter sensiblement les performances au détriment d'un peu de généricité. Les résultats chiffrés de nos expériences sont présentés et examinés dans la section 6.

2 Genericité en traitement d'images

Un framework logiciel générique fournit des algorithmes et des structures de données orthogonaux disposant chacun d'une *unique* implémentation, pouvant être utilisés ensembles pour réaliser toute combinaison valide. Cette approche permet d'éviter la redondance de code et favorise une réelle réutilisabilité des algorithmes sur une multitude de structures de données compatibles (par exemple, des types d'images), tout en évitant l'explosion combinatoire.

La généricité est basée sur le paradigme de programmation générique (PG). L'idée maîtresse de cette approche est de construire le logiciel à l'aide de *concepts*, qui représentent des entités abstraites du domaine (ici, le TDI) [6]. Un concept définit les relations entre l'entité qui lui est associée (par exemple un type d'images) et d'autre éléments (par exemple, un type de points ou un type de valeurs). Il comporte également l'ensemble minimal des services à fournir (par exemple, l'obtention d'une valeur associée à un point dans une image). On peut par la suite écrire des algorithmes génériques en utilisant les concepts à la place des types de données spécifiques. Comme ces algorithmes ne font pas état des détails liés aux types de données manipulés, ils constituent des implémentations génériques, ne dépendant d'aucun type d'entrées en particulier. Nous avons utilisé cette approche avec succès en morphologie mathématique [5] et dans d'autre domaines du TDI.

Nous pouvons illustrer cette idée à l'aide d'un algorithme élémentaire : une dilatation morphologique utilisant un élément structurant plat. L'algorithme 1 montre une implémentation simple de ce filtre. Cependant, celle-ci comporte des détails d'implémentation qui lient la routine à un type d'entrées spécifique (une image 2D sur une grille régulière contenant des valeurs compatibles avec le type `unsigned char`). De plus, l'élément structurant (4-connexe) ne peut être changé. Par conséquent, il nous est impossible d'utiliser cet algorithme pour traiter, par exemple, une image 3D en couleurs (RVB) avec un élément structurant 6-connexe.

Cependant, si un algorithme fait usage d'entités abstraites basées sur des concepts, il peut être employé avec des types d'entrées variés. C'est le cas de l'algorithme 2, qui propose une version générique de l'algorithme précédent, implémenté à l'aide d'une bibliothèque C++ générique, Milena, pièce centrale d'une plate-forme libre de TDI, Olena [7]. Les types d'entrées (resp. une image, et un élément structurant ou *fenêtre*) sont désormais des paramètres de l'algorithme (resp. `I` et `W`) ; les boucles sur les intervalles de lignes et colonnes sont remplacées par un unique objet `p` parcourant le domaine de l'image, appelé *itérateur sur sites* (ou *piter*) ; de même, les points de l'élément structurant (auparavant figé dans le code) centré en `p` ne sont plus mentionnés explicitement et sont remplacés par un second itérateur `q` (nommé *qiter*) parcourant la fenêtre `win` ; enfin, en lieu et place du calcul manuel d'une valeur maximale, un objet *accumulateur* est utilisé pour calculer le supremum des valeurs dans la fenêtre glissante.

3 Généricité et performance

Les chiffres de la table 1 font apparaître un surcoût à l'exécution pour l'implémentation générique (algorithme 2), environ dix fois plus lente que la version non générique (algorithme 1). Il ne s'agit pas d'une conséquence du paradigme de PG en soit. Ce surcoût est en fait dû au style hautement abstrait de l'algorithme 2, qui en retour rend la routine très polyvalente vis-à-vis du contexte d'utilisation. La version non générique est plus rapide que son homologue générique car elle tire parti de caracté-

ristiques connues de ses entrées. Par exemple, l'élément structurant est « intégré » dans la fonction (alors qu'il s'agit d'un objet représentant une entrée générique dans l'algorithme 2) : sa taille est constante et connue à la compilation. De tels détails d'implémentation sont des informations *statiques* que le compilateur peut utiliser pour optimiser le code produit. La raison qui empêche un code d'être générique apparaît donc comme une condition nécessaire à la génération de code efficace : les détails d'implémentation.

4 Optimisations génériques

En choisissant soigneusement la quantité d'information spécifique apparaissant dans un algorithme, il est possible de créer des variantes intermédiaires affichant de bonnes performances à l'exécution et préservant un grand nombre de traits génériques. Une technique permettant d'accélérer l'algorithme 2 consiste par exemple à ne pas employer d'itérateurs sur sites pour parcourir le domaine des images d'entrée et de sortie. Dans Milena, un itérateur sur site sait se convertir automatiquement en un site (point), c'est-à-dire une position dans le domaine d'une (ou plusieurs) image(s). Une telle information de position n'est pas liée à une image en particulier : dans le cas des images 2D régulières, un site `point2d(42, 51)` est compatible avec tout domaine 2D à coordonnées entières de la bibliothèque (ce qui inclut des espaces toriques, des sous-espaces non rectangulaires de $\mathbb{Z}^2$, etc.). C'est pour cette raison que l'itérateur `p` est utilisé pour désigner le même emplacement à la fois dans `input` et `output` dans l'algorithme 2.

L'utilisation d'une expression aussi générale se traduit habituellement par un coût à l'exécution : chaque utilisation de l'itérateur avec une image implique des calculs, car cet itérateur ne pointe pas directement sur les données. Cette souplesse n'est cependant pas toujours nécessaire, si les données à traiter présentent des propriétés particulières. Par exemple, une image dont les valeurs sont stockées dans un espace mémoire contigu et linéaire peut être parcourue en utilisant un pointeur. Ce dernier permet un accès direct aux valeurs de façon séquentielle, en utilisant leurs adresses mémoire au lieu de calculer ces emplacements à chaque accès. Dans Milena, nous encapsulons ces pointeurs dans des objets légers appelé *itérateurs sur pixels* ou *pixters* (un pixel désignant un couple (site, valeur) dans une image). Un pixter est lié à une seule image et ne peut être utilisé pour itérer sur une autre image. Les pixters peuvent également être employés pour parcourir des éléments structurants (fenêtres) spatialement invariants, à condition que le domaine sous-jacent de l'image soit régulier.

L'algorithme 3 est une réimplémentation de l'algorithme 2 où les itérateurs sur sites sont remplacés par des itérateurs sur pixels. Le code est similaire, à ceci près que les images `input` et `output` sont maintenant parcourues à l'aide de deux itérateurs différents (possédant chacun un pointeur sur les données de l'image correspondante). Cette implémentation de dilatation morphologique est moins générique que algorithme 2. Malgré

```
template <typename I, typename W>
I dilation(const I& input, const W& win) {
  I output; initialize(output, input);
  // Itérateur sur les pixels de 'input'.
  mln_pixter(const I) pi(input);
  // Itérateur sur les pixels de 'output'.
  mln_pixter(I) po(output);
  // Itérateur sur les pixels voisins de 'pi'.
  mln_qixter(const I, W) q(pi, win);
  for_all_2(pi, po) { // Itération synchrone de 'pi' & 'po'.
    accu::supremum<mln_value(I)> sup;
    for_all(q)
      sup.take(q.val());
    po.val() = sup.to_result();
  }
  return output;
}
```

ALGORITHME 3 – Dilatation optimisée, en partie générique

cela, elle est toujours utilisable avec une grande variété de types d'images, du moment que leurs données présentent une organisation régulière, ce qui inclut les images régulières de dimension quelconque dont les valeurs sont stockées sous forme d'un unique bloc de mémoire linéaire. De plus, cette implémentation est compatible avec n'importe quel élément structurant spatialement invariant (c'est-à-dire toute fenêtre constante). Cette variante reste par conséquent plus générique que l'algorithme 1. Du point de vue des performances, l'algorithme 3 est comparable à l'algorithme 1 (voir la table 1). Il s'agit donc d'une bonne alternative à la dilatation générique, où l'équilibre entre la performance et la généricité est déplacé vers la première.

L'approche présentée ici s'applique aussi à d'autres algorithmes de la littérature du TDI pour lesquels des implémentations optimisées ont été proposées. De telles optimisations sont en pratique le plus souvent compatibles avec une variété de types d'entrées ; leurs implémentations peuvent donc être considérées comme des *optimisations génériques* puisque qu'elles ne sont pas liées à un type spécifique.

5 Optimisations supplémentaires

Il est possible de développer l'approche proposée dans cet article afin d'améliorer les performances d'une optimisation générique. L'idée est de faire participer les structures de données à cet effort d'optimisation : au lieu d'intervenir uniquement sur les algorithmes, nous pouvons produire de nouvelles variantes optimisées en agissant également sur leurs entrées.

Par exemple, au lieu d'une fenêtre contenant un tableau dynamique de vecteurs (à savoir $\{(-1,0),(0,-1),(0,0),(0,+1),(+1,0)\}$ dans le cas d'un élément structurant 4-connexe spatialement invariant) – dont la taille est connue à l'exécution – nous pouvons implémenter et utiliser une fenêtre *statique*, composée d'un tableau comportant les mêmes informations, mais dont le contenu et la taille sont connus à la compilation. Les compilateurs modernes savent tirer parti de telles informations pour effectuer des optimisations efficaces (par exemple remplacer une boucle sur les éléments de la fenêtre par un code déroulé équivalent). Dans cet exemple en particulier, cette optimisation nécessite juste la création de deux types de données

TABLE 1 – Temps d'exécutions des algorithmes de dilatation

Implémentation	Temps (secondes)		
Image (pixels) :	512^2	1024^2	2048^2
Non générique (Alg. 1)	0.10	0.39	1.53
Non générique, avec pointeurs[2]	0.07	0.33	1.27
Générique (Alg. 2)	0.99	4.07	16.23
Optimisée en partie gén. (Alg. 3)	0.13	0.54	1.95
Alg. 3 avec une fenêtre statique	0.06	0.28	1.03

relativement simples (fenêtre et itérateur sur pixels statiques). Aucune nouvelle implémentation de l'algorithme de dilatation n'est nécessaire : il suffit d'utiliser l'algorithme 3 avec le nouveau type de fenêtre. Le code résultant est non seulement plus rapide que la version non générique de l'algorithme 1, mais peut aussi être plus rapide qu'une version optimisée manuellement (et par conséquent non générique) utilisant des pointeurs (voir la section suivante).

6 Résultats

La table 1 montre les temps d'exécutions de plusieurs implémentation de dilatation morphologique avec un élément structurant (fenêtre) 4-connexe, appliquée à des images de taille croissante (512×512, 1024×1024 et 2048×2048 pixels). Les temps correspondent à 10 invocations itératives. Les tests ont été effectués sur un PC sous Debian GNU/Linux comportant un processeur Intel Pentium 4 à 3,4 GHz et 2 Go de mémoire vive à 400 MHz, en utilisant la version 4.4.5 du compilateur `g++` (GCC), appelé avec le niveau d'optimisation '`-O3`'. En sus des implémentation présentées dans cet article, une version supplémentaire non générique utilisant des optimisations à base de pointeurs a été ajoutée à la suite de tests, afin de pousser plus encore la comparaison entre du code non générique – optimisé essentiellement « à la main » – et du code générique – optimisé essentiellement par le compilateur.

Le coût lié à l'abstraction dans l'implémentation la plus générique est important : il est environ dix fois plus long que l'algorithme 1. Le code particulièrement flexible de l'algorithme 2 est dépourvu de détail d'implémentation que le compilateur pourrait utiliser pour produire du code rapide. L'algorithme 3 propose un compromis entre généricité et performance : il est environ 30% plus lent que l'algorithme 1, mais il est suffisamment générique pour fonctionner avec de nombreux types d'images régulières (qui sont dans les faits les plus courants). Le cas de la dilatation avec une fenêtre statique est intéressant : la réutilisation d'un même code (algorithme 3) avec une entrée moins générique (une fenêtre statique représentant un élément structurant fixe spatialement invariant) génère un code deux fois plus rapide, au point de surpasser une implémentation optimisée manuellement à base de pointeurs. Ainsi, il est utile de disposer de plusieurs implémentations (ici les algorithmes 2 et 3) pour satisfaire des besoins de flexibilité et d'efficacité.

2. Cette implémentation n'est pas montrée ici pour des raisons de place.

7 Conclusion

Cet article propose une approche pour concilier généricité et performance dans le logiciel pour le TDI. Cette stratégie repose sur la notion d'optimisation générique, qui exprime une variante efficace d'un algorithme pour un sous-ensemble des types d'entrées valides.

L'ajout de versions moins génériques mais plus performantes d'algorithmes ne devrait pas affecter la motivation à concevoir un framework de TDI aussi générique que possible. Nous pensons que la version la plus générique d'un algorithme devrait toujours être écrite en premier, puis complétée par des implémentations plus rapides. En effet, disposer d'une version générique d'un algorithme signifie posséder au moins une implémentation fonctionnant avec tous les types d'entrées valides. De plus, les implémentations génériques sont généralement plus simples, plus courtes et plus rapides à écrire, si le framework utilisé fournit les entités nécessaires (itérateurs, fenêtres, etc.). Enfin, elles constituent une base saine pour l'écriture de variantes optimisées, de par la similarité de leurs structures.

Il est également possible d'automatiser la sélection de la version d'un algorithme en fonction du type des entrées utilisées. Ce mécanisme nécessite d'équiper les types de données de propriétés (*traits*) décrivant leurs capacités. Les techniques de métaprogrammation C++ permettent alors l'écriture d'un code de sélection statique basé sur ces propriétés.

La bibliothèque Milena, utilisée dans les exemples de cet article, est distribuée avec la plate-forme Olena. Ce projet logiciel libre, diffusé sous licence GNU GPL, est accessible à l'adresse `http://olena.lrde.epita.fr/Download`.

Références

[1] Adobe. Generic Image Library (GIL). `http://opensource.adobe.com/gil`.

[2] Centre de Morphologie Mathématique. Morph-M. `http://cmm.ensmp.fr/Morph-M/`.

[3] L. Ibáñez, W. Schroeder, L. Ng et J. Cates. *The ITK Software Guide*. Kitware, Inc., 2005.

[4] U. Köthe. Reusable software in computer vision. *In* B. Jähne, H. Haussecker et P. Geißler, éditeurs. *Handbook of Computer Vision and Applications*, volume 3. Academic Press, 1999.

[5] R. Levillain, Th. Géraud et L. Najman. Milena : Write generic morphological algorithms once, run on many kinds of images. *In* M. H. F. Wilkinson et J. B. T. M. Roerdink, éditeurs. *Proc. of the 9th International Symposium on Mathematical Morphology*, Groningen, The Netherlands, 2009.

[6] R. Levillain, Th. Géraud et L. Najman. Why and how to design a generic and efficient image processing framework : The case of the Milena library. *In Proc. of the International Conference on Image Processing*, Hong Kong, 2010.

[7] LRDE. The Olena image processing platform. `http://olena.lrde.epita.fr`.