NT2 : Une bibliothèque haute-performance pour la vision artificielle

NT2 : A High-performance library for computer vision.

Joël Falcou Jean-Thierry Lapresté Thierry Chateau Jocelyn Sérot

{prenom.nom@lasmea.univ-bpclermont.fr}

LASMEA - CNRS UMR 6602
Campus des Cézeaux 63000 AUBIERE

Résumé

Cet article présente NT2, une solution logicielle simplifiant la mise au point d'applications de vision artificielle complexes tout en garantissant un haut niveau de performance. Pour ce faire, la bibliothèque NT2 propose une interface de haut niveau – proche de celle de MATLAB® *– et offre des performances proches de celles obtenues en C grâce à des techniques d'implantation avancées comme l'évaluation partielle et la méta-programmation* template. *Nous démontrons la pertinence de cette solution en présentant les performances de plusieurs noyaux d'applications classiques de traitement d'images et de vision artificielle.*

Mots Clef

méta-programmation *template,* évaluation partielle, MATLAB.

Abstract

This paper introduces NT2, a software framework that eases the development of complex computer vision applications while maintaining high level performance. To do so, the NT2 library proposes a high-level, close to MATLAB *interface and allows execution times to be on par with a hand-made C implementation by using advanced software enginering technics like partial evaluation and* template *meta-programming. We further demonstrates NT2 impact with the analysis of some widely used computer vision algorithms and image processing performances.*

Keywords

template meta-programming, partial evaluation, MATLAB.

1 Introduction

Les applications de vision artificielle complexes sont habituellement développées à partir d'un formalisme mathématique adéquat et implantées dans un langage de haut-niveau qui prend en charge les éléments classiques du calcul matriciel, de l'algèbre linéaire ou du traitement d'image. Des outils comme MATLAB [18] sont régulièrement utilisés par les développeurs de la communauté Vision car ils permettent d'exprimer de manière directe des expressions mathématiques complexes et proposent un large panel de fonctionnalités au sein d'un modèle de programmation procédural classique. Mais il s'avère que certaines applications nécessitent d'être portées vers des langages plus performants en terme de temps de calcul.

Pour ce faire, un processus de réécriture vers des langages comme C ou FORTRAN est souvent envisagé mais implique un surcroît de travail non-négligeable compte tenu du faible niveau d'expressivité de ces langages. Une alternative consiste à utiliser un langage comme le C++ que ses fonctionnalités rendent *a priori* intéressant dans le cadre du calcul scientifique. Ainsi, la surcharge des opérateurs, qui augmente l'expressivité du langage et son aspect orienté-objet qui favorise l'abstraction et la réutilisation de code sont autant d'atouts qui facilitent cette transition.

Malgré ces fonctionnalités, le C++ est peu utilisé au sein d'applications de Vision car ses performances sont incompatibles avec leurs contraintes temporelles. En effet, les programmes C++ sont beaucoup plus lent que leurs équivalents C ou FORTRAN – d'un facteur compris entre 1.2 et 10 – ce qui force un grand nombre de développeurs à écrire en C ou en FORTRAN les parties critiques de leurs algorithmes, n'utilisant le C++ que pour des tâches annexes comme les entrées-sorties ou la gestion des interfaces graphiques et ce malgré l'existence de bibliothèques comme OpenCV [12],VXL ou Gandalf.

Le développeur d'applications de vision artificielle se retrouve donc à devoir effectuer un choix entre l'expressivité de ses outils et leur efficacité. Quand bien même il est possible de définir de nouveaux outils proposant une expressivité plus grande, leurs techniques d'implantations classiques dégradent rapidement leur efficacité (cf. fig. 1).

Cet article présente une nouvelle approche qui allie les

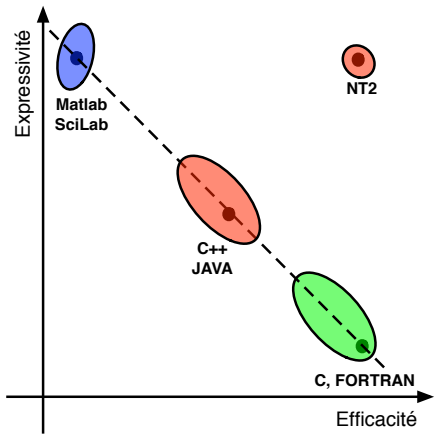

FIG. 1 – *Tension expressivité-efficacité des langages usuellement employés pour le développement d'applications de vision artificielle. Chaque langage permet d'exprimer un ensemble d'outils de plus haut-niveau (représentés ici par les zones colorées) qui, en général, augmente l'expressivité au dépend de l'efficacité. NT2 propose de définir une nouvelle méthode d'implantation permettant de s'affranchir de cette limitation.*

performances de langages comme C et FORTRAN et l'expressivité de MATLAB en tirant parti du haut-niveau d'abstraction des langages orientés objet. Cette solution prend la forme d'une bibliothèque C++ – la bibliothèque NT2 – qui permet de simplifier le portage d'application MATLAB en C++ tout en introduisant un surcoût minime par rapport à une version C.

Nous présenterons d'abord les limitations des implantations classiques des bibliothèques de calcul scientifique en C++, puis nous détaillerons les solutions de développement utilisées par NT2. Nous présenterons ensuite les fonctionnalités de NT2 avant d'évaluer ses performances tant en terme d'efficacité qu'en terme d'expressivité sur des noyaux d'applications classiques de vision par ordinateur.

2 Le calcul scientifique en C++

L'écriture de bibliothèques dédiées au calcul scientifique en général et à la vision par ordinateur en particulier a fait l'objet de nombreuses tentatives. Ces travaux démontrent que l'expressivité du langage peut être relevée à un très haut-niveau mais que les techniques d'implantations classiques de ces solutions entrainent une perte sévère de performances. Nous allons dans un premier temps essayer de cerner la source de ces pertes puis proposer un schéma d'implantation plus adapté qui permettra de conserver des performances élevées sans sacrifier l'expressivité de nos outils.

2.1 Problématiques

La manipulation de vecteurs, matrices et tableaux sont à la base de la plupart des codes de calcul scientifique. En C++, il est possible de définir des classes qui encapsulent ce type d'objet et qui fournissent **des opérateurs surchargés** – comme $+, -, \times$ ou $/$. Ce mécanisme de surcharge permet aux bibliothèques de calcul à base de vecteurs et de matrices de proposer une interface naturelle. Ainsi, la somme de trois vecteurs s'écrit simplement :

$$w = x + y + z$$

Mais, lorsque le compilateur traite une série d'appel d'opérateurs surchargés, il procède à une évaluation successive de chaque opérande et génère un objet temporaire. Cette évaluation dyadique force le compilateur à évaluer tout d'abord $t_1 = x + y$, puis $t_2 = t_1 + z$ pour enfin affecter t_2 à w, t_1 et t_2 étant des objets temporaires crées uniquement pour servir de stockage aux résultats intermédiaires.

L'évaluation dyadique des opérateurs surchargés est lente car, au lieu d'évaluer une expression via une seule boucle, plusieurs boucles de traitements et vecteurs temporaires sont nécessaires. Ces vecteurs temporaires sont généralement alloués dynamiquement, causant une perte de performance notable pour des tableaux de petite taille. Par exemple, l'évaluation de $w = x + y + z$, où x, y et z contiennent quatre nombres réels simple précision, ne demande que quelques cycles alors que l'allocation et la libération des tableaux temporaires en demande près d'une centaine, rendant la manipulation de tableaux de petite taille en C++ extrêmement peu efficace. Pour des matrices ou des vecteurs de grande taille, la principale perte de performance provient de la bande passante de la mémoire. En effet, de nombreux codes de calcul sont limités non pas par le temps de calcul effectif mais par le temps de transfert entre la mémoire principale et le processeur. Dans notre exemple, si chaque tableau contient 1Mo de données, la version à base d'opérateurs surchargés nécessite 8Mo de transfert entre le processeur et la mémoire, soit deux fois la quantité effectivement nécessaire. On montre alors que, en moyenne, l'évaluation d'une expression de M opérandes et N opérateurs s'exécute $\frac{2N}{M}$ fois moins vite que le code équivalent en C ou en FORTRAN [25].

Une autre source de perte de performances réside dans l'utilisation de classes polymorphes[1] pour encapsuler des comportements variés. On utilise souvent cette technique pour fournir des classes de vecteurs ou de matrices proposant des optimisations diverses comme par exemple le choix de l'index de départ des accès, la gestion de la forme de la matrice (diagonale, bande ou triangulaire, etc.). Ce type d'option s'implémente en général par l'intermédiaire de **méthodes virtuelles** que chaque variante de la classe de base va surcharger. Or, l'appel d'une méthode virtuelle nécessite un accès mémoire supplémentaire afin de résoudre l'indirection des pointeurs de fonctions de la *vtable* [2] de la classe. Ce surcoût est relativement négligeable lorsque le

[1] c'est à dire faisant partie d'une hiérarchie de classe.
[2] ou table des méthodes virtuelles

temps d'exécution de la méthode virtuelle elle-même est suffisamment important. Malheureusement, dans le cas des options habituellement supportées par des classes de matrices, ce temps d'exécution est extrêmement faible.

2.2 Les bibliothèques actives

On pourrait croire que des phases d'optimisations intensives ou l'utilisation de techniques de programmation générique classiques telles que celles mises en œuvre dans les bibliothèques VIGRA[15] ou ITK peuvent éliminer ces problèmes. En général, il n'en est rien car les compilateurs n'ont pas accès à la sémantique des abstractions mises en œuvres. Là où un développeur voit un ensemble d'opérations sur des tableaux numériques, le compilateur ne voit que des boucles et des allocations mémoires. L'utilisation des *templates* de manière naïve élimine certes les surcoûts dus aux appels de méthodes ou au polymorphisme dynamique mais ne permet pas – du moins dans le type de solution classiquement proposé – d'effectuer des optimisations inter-appels.

Une approche plus efficace est de développer des bibliothèques qui proposent à la fois un niveau d'abstraction élevé et un mécanisme d'optimisation adapté utilisant non plus seulement une programmation générique à base de *templates* mais bien une **programmation générative** capable de générer un code optimisé en fonction d'un ensemble de spécifications statiques. Ce concept de «bibliothèque active» [24] met en œuvre une phase d'optimisation haut-niveau dépendant du modèle de programmation, laissant le compilateur effectuer les optimisations bas-niveau usuelles (allocation des registres et ordonnancement des instructions par exemple). Pour réaliser de telles librairies, plusieurs systèmes de méta-définition ont été proposés. Parmi ceux ci, on peut citer des projets comme Xroma [4], MPC++ [13], Open C++ [3] ou Magik [5]. Ces systèmes permettent d'injecter au sein d'un compilateur C++ des extensions qui permettent de fournir à ce dernier les indications sémantiques nécessaires à une compilation efficace. L'utilisation de ces outils reste néanmoins difficile et contraint les développeurs à maitriser des outils annexes potentiellement complexes. Il est donc nécessaire de chercher une alternative intégrable directement au sein du langage C++ sans nécessiter d'outils externes.

2.3 Les *Expression Templates*

L'idée maîtresse de NT2 repose sur l'utilisation d'un idiome appelé *Expression Templates* [22] qui consiste à déterminer quels aspects d'une expression mathématique peuvent être évalués lors de la compilation. Cette technique se base sur les propriétés intrinsèques des *templates* C++ qui se comportent comme un langage Turing-complet [21, 23].

Considérons par exemple une expression mettant en jeu des opérateurs manipulant une classe de tableaux numériques :

$$r = a + b + c$$

Cette expression s'évalue en trois passes : évaluation de $a+b$, de la somme de ce résultat intermédiaire avec c et finalement recopie du résultat final dans r.

$$\begin{aligned}&\forall i, t_1[i] = a[i] + b[i]\\ &\forall i, t_2[i] = t_1[i] + c[i]\\ &\forall i, r[i] = t_2[i]\end{aligned}$$

Le but est alors de se ramener à une évaluation **en une seule passe** :

$$\forall i, r[i] = a[i] + b[i] + c[i]$$

Pour cela, il faut s'intéresser à la **structure** de l'expression à droite de l'affectation. La structure de l'arbre de syntaxe abstraite associé à cette expression est clairement fixée au moment de l'écriture de cette expression, alors que le contenu des tableaux n'est connu qu'à l'exécution. Cette dichotomie montre que, dans l'évaluation d'une telle expression, la composition des opérateurs est une donnée *statique* et que les données contenues dans les tableaux sont *dynamiques*. L'application d'un évaluateur partiel à une expression comprenant M opérandes et N opérateurs revient à créer lors de la compilation une fonction φ à M arguments effectuant l'application des N opérateurs de l'expression initiale sur une unique valeur scalaire. A l'exécution, seule l'application de cette fonction composite aux données initiales sera évaluée. Ici, nous obtenons :

$$\varphi(x, y, z) = x + y + z$$

$$\forall i, r[i] = \varphi(a[i], b[i], c[i])$$

L'idée est donc de définir un certain nombre de classes permettant de représenter **au sein du langage** la structure même de l'arbre de syntaxe abstraite afin de pouvoir «construire» φ au moment de la compilation. Pour cela, nous définissons plusieurs classes *templates* qui vont nous permettre de représenter respectivement la structure de l'arbre, ses éléments terminaux et les opérations effectuées à la traversée des nœuds de l'arbre. L'ensemble de ces classes va alors évaluer partiellement l'expression à la compilation et générer un code résiduel correspondant à la suite d'opérations strictement nécessaires pour une évaluation dynamique optimisée [22, 8].

2.4 Gestion statique du polymorphisme

En ce qui concerne la gestion des comportements optionnels, une solution consiste à utiliser une version *template* du patron de conception *Strategy* [10]. Ce patron définit une famille d'algorithmes encapsulés et interchangeables. *Strategy* permet de modifier l'algorithme sans toucher au code qui l'utilise et réciproquement. Dans notre cas, cette famille d'algorithmes est implantée comme une série de structures *templates* [1, 2] exposant une ou plusieurs méthodes de classe qui seront utilisées au sein même

des méthodes des classes de matrices et de tableaux. De part leur nature *template,* ces méthodes sont directement injectées à l'endroit même de leur appel, éliminant ainsi les cycles nécessaires pour déréférencer une adresse de méthode virtuelle.

3 La bibliothèque NT2

NT2 [3] [7] est une bibliothèques de classes C++ permettant la gestion aisée et performante, de matrices numériques. Son originalité réside principalement en trois points :

– L'utilisation de techniques à base d'*Expression Templates* permettant d'éviter des recopies de données et rapprochant les performance du code obtenu de celles d'un code C écrit à la main grâce au moteur de génération de code E.V.E. [8] ;

– L'utilisation transparente de primitives SIMD présentes sur une large gamme de processeurs ;

– La reproduction la plus fidèle possible des fonctionnalités propres à MATLAB.

Parmi ces trois points, nous nous focalisons sur l'aspect expressivité de NT2 en comparant son interface à celle de MATLAB sans nous attarder sur l'aspect implantation déjà décrit en détail dans [8].

3.1 Fonctionnalités propres à MATLAB

MATLAB fournit un certain nombre de constructions qui permettent de définir et manipuler de manière aisée des tableaux numériques et des matrices. NT2 se propose de fournir des écritures le plus proche possible pour les opérations de base au sein de la syntaxe C++.

Création et assemblage de tableaux. NT2 propose plusieurs modalités de définitions de tableaux et de matrices. La principale différence avec MATLAB réside bien entendu dans la nécessité de **typer** ses variables et la possibilité d'expliciter la taille des matrices et tableaux au moment de leur création via la fonction `ofSize`.

Listing 1 – Construction de matrices NT2

```
// matrix vide
matrix<int> a;

// Matrice nulle 4x4
matrix<double> b(ofSize(4,4));

// Matrice identité 3x3
matrix<double> c(eye(3));

// Copie de c dans d
matrix<double> d(c);
```

NT2 supporte aussi les créations de tableaux par énumération totale ou partielle des éléments d'un tableau qui se retrouvent sous la forme des fonctions `iota` et `cons`. De même, les fonctions MATLAB comme `ones`, `zeros`, `eye`, `rand` et `linspace` sont disponible telles quelles. Enfin, les opérations de concaténation de tableaux s'exprime via les fonctions `cath` et `catv`. Le tableau 1 résume les correspondances entre les écritures MATLAB et leur équivalents NT2.

MATLAB	NT2
`a = 3:2:13`	`a = iota(3,2,13);`
`b = [1 2 3 4]`	`b = cons(1,2,3,4);`
`c = eye(4)`	`c = eye(4);`
`d = [a b]`	`d = cath(a,b);`
`e = [c;c]`	`e = catv(d,c);`

TAB. 1 – *Exemples de correspondance MATLAB/NT2 pour la construction de conteneurs.*

Accès aux éléments des tableaux. NT2 fournit un équivalent à l'ensemble des méthodes d'accès aux éléments d'un tableau ou l'extraction d'un sous-tableau. Néanmoins, pour des raisons de syntaxes, les opérateurs comme : ou `1:N` sont transformés en écritures compatibles avec la syntaxe du C++ comme indiqué dans le tableau 2. En outre, NT2 supporte de manière naturelle l'interpolation bilinéaire lors d'un accès direct utilisant des index réels là où MATLAB nécessite d'expliciter cette interpolation.

MATLAB	NT2
`a(:)`	`a(All())`
`a(1:2:10)`	`a(Range(0,2,9))`
`a(1:end)`	`a(Begin(),End())`
`a(b > 3)`	`a(b > 3)`

TAB. 2 – *Exemples de correspondance MATLAB/NT2 pour l'accès aux éléments d'un conteneur.*

Opérations et fonctions. L'ensemble des fonctions arithmétiques, logiques et trigonométriques usuelles est supporté et utilise une syntaxe identique à celle de MATLAB. Seules exceptions notable, la différentiation faites par MATLAB entre `*` et `.*` se transcrit en C++ par l'ajout d'une fonction `mul` qui se substitue à l'écriture `.*`, `*` conservant son sens habituel de produit matriciel et la fonction `trans` qui remplace l'écriture `'` pour la transposée de matrice.

NT2 reproduit d'ailleurs une partie des fonctionnalités d'algèbre linéaire de MATLAB en s'appuyant sur la librairie LAPACK [4]. Ces fonctionnalités comprennent :

– Les fonctions `mldivide` et `mrdivide` équivalentes aux opérateurs de division gauche et division droite de MATLAB. Les algorithmes utilisés pour déterminer la

[3] *Numerical Template Toolboxes*

[4] Ce ne doit pas être surprenant puisque c'est aussi le choix de base actuel de MATLAB.

méthode de résolution adéquate sont, autant que faire ce peut, identique à ceux de MATLAB, une méthode spécifique permettant d'accéder *a posteriori* à la méthode utilisée effectivement pour résoudre un système donné ;

– Des classes de décompositions qui incluent des méthodes de résolutions et des méthodes utilitaires variées (conditionnement, trace, déterminant, etc.) pour des algorithmes comme QR, PQR, PLU et SVD.

3.2 Fonctionnalités supplémentaires

NT2 propose en outre un certain nombre de fonctionnalités propre qui permettent, une fois le code porté depuis MATLAB, d'effectuer des optimisations supplémentaires.

Conteneurs optimisés. MATLAB propose un unique type de conteneur pour lequel aucune sémantique forte n'est vraiment définie. Afin de pouvoir optimiser un certain nombre d'écritures, NT2 définit une famille de conteneur inter-compatible spécifiant chacun des comportements différents :

– `array` et `matrix` se basent sur la sémantique classique des tableaux numériques, `matrix` étant optimisé pour le traitement des tableaux à une ou deux dimensions.

– `view` et `stencil` permettent respectivement de gérer de manière transparente des vues contigües sur un espace mémoire pré-existant et des vues potentiellement non-contigües sur un conteneur NT2 pré-existant. Classiquement, `view` permet d'utiliser sans recopie un tableau C issue d'une allocation libre (pour manipuler par exemple l'espace mémoire contenant une image issue d'un *driver* caméra). `stencil` permet quant à lui de travailler sur des sous-tableaux extraits sans copies d'un conteneur NT2 (pour travailler par exemple sur un sous-échantillonnage d'une image).

– `rigid`, `projection` et `homography` permettent de gérer de manière naturelle les transformations rigides, les projections et les homographies 2D et 3D. Leur intérêt est de proposer un ensemble de méthodes supplémentaires correspondant à leur nature géométrique (comme la construction d'une homographie à partir de la mise en correspondance de deux plans par exemple). Enfin, l'application de ces transformations géométriques à un point ou un ensemble de point passe simplement par l'opérateur `*`.

– `polynome` qui encapsule les méthodes classiques de constructions (par racines ou coefficients) des polynômes, leur évaluation, l'extraction de leur racines, de leur degré et de leur valuation.

Boîtes à outils annexes. NT2 propose une implantation d'algorithmes plus généraux :

– Interpolation polynomiale et par spline.

– Optimisation non-linéaire via les méthodes de Brent et Levenberg-Marquardt [16, 17] ;

ainsi que des algorithmes plus orientés «vision artificielle» :

– Détection de contour et de points d'intérêts par les méthodes de Harris et Stephen [11] ou Rosten et Drummond [19] ;

– Un algorithme de *fast marching* [20].

Ces outils s'appuient sur les performances des fonctions de bases de NT2 pour fournir des implantations dont les performances, bien qu'inférieures à celles d'implantation optimisées, restent très satisfaisantes.

Options spécifiques. Afin de permettre de régler finement les performances de l'application finale, NT2 permet de spécifier un certain nombre d'options à la création de ces conteneurs en s'inspirant du modèle de programmation de HPF [14]. On trouve dons des options permettant :

– **L'activation des optimisations SIMD**.
De nombreux facteurs comme le type de données et le type de fonctions utilisés vont modifier le type d'optimisation applicable au moment de l'évaluation des expressions mettant en jeu des instances de conteneur. L'utilisation du marqueur `simd` va indiquer au noyau d'optimisation *template* que les expressions utilisant ces instances pourront utiliser les optimisations à base de primitives SIMD présente sur la plateforme courante (ALTIVEC ou SSE2).

– **L'activation d'un déroulage des calculs**.
Lors de l'évaluation d'une expression, il est possible de forcer NT2 à dérouler la boucle d'évaluation d'expressions d'un facteur défini par l'utilisateur. Ce déroulage permet dans certains cas de remplir de manière optimale le cache du processeur et d'augmenter les performances du calcul en cours.

– **Une stratégie d'allocation mémoire**.
L'allocation de la mémoire d'un conteneur peut être dynamique, statique ou être déléguée à une classe définie par l'utilisateur. Par défaut, cette allocation mémoire est effectuée de manière dynamique.

– **L'indexation de la matrice**.
Chaque conteneur peut définir sa base d'indexation. Par défaut, l'indexation des éléments d'une de ces instances commence à 0 comme en C et l'utilisation d'une base d'indexation égale à 1 permet de simuler le comportement classique des tableaux MATLAB.

Le listing 2 donne un exemple de déclarations utilisant diverses combinaisons de marqueurs.

Listing 2 – Exemples d'instanciation de conteneur NT2

```
// Vue indexée Matlab de d:
view<complex<float>,settings<base_index<1> > > b(d);

// Matrice allouant statiquement 20 octets:
matrix<float,settings<static_storage<20> > > c;

// Polynôme utilisant un déroulage de 2 et le SIMD
polynome<float,settings<unroll<2>,simd > > e;
```

Ces marqueurs sont passés en arguments aux classes de conteneurs de NT2 via la méta-fonction `settings`. On notera que l'ordre des marqueurs n'est pas imposé. En outre, si des options incompatibles avec le type de conteneur sont spécifiées, un message d'erreur sera émis lors de la compilation.

Afin de simplifier le développement d'applications et de pas noyer l'utilisateur final sous une multitude de déclarations utilisant des listes de marqueurs diverses et variées, NT2 fournit une fonction `view_as` dont le paramètre *template* permet de spécifier à la volée de nouveaux marqueurs sur des instances existantes d'un conteneur quelconque sans effectuer de recopie.

4 Performance

Nous allons nous intéresser maintenant à quantifier les deux aspects de NT2 : son **efficacité**, c'est-à-dire ses performances temporelles comparées à celle d'un code équivalent en C et son **expressivité**, en comparant sa facilité d'accès en comparant le code MATLAB initial de plusieurs algorithmes de traitements d'images et de vision avec leur version C++. Nous attardons plus spécifiquement à spécifier les performances des fonctions de base de NT2, sur un algorithme de traitement d'images et des algorithmes de vision artificielle.

4.1 Fonctions de base

Ces tests consistent à mesurer les temps d'exécutions d'un code MATLAB , C et C++ utilisant NT2 effectuant des calculs simple (addition, produit matricielle et somme des éléments d'un tableaux) sur des tableaux de grandes tailles (de l'ordre de 1Mo de réels simple précision) afin d'évaluer les gains apportés par NT2 par rapport à MATLAB (Γ_M) et l'efficacité de NT2 vis à vis du langage C en terme de temps de calcul (ε_C). L'ensemble de ces test ont été effectués sur une machine Intel Pentium 4 à 1.8GHz[5] et leurs résultats sont présentés dans le tableau 3.

Les temps d'exécution pour les opérations régulières (addition, somme et autres opérations arithmétiques) sont très proche de ceux obtenus en C (de l'ordre de 75 à 90%) et sont en moyenne quatre à cinq fois plus rapide que

[5]Des résultats identiques ont été obtenus sur un PowerPC G5 à 2Ghz

Opration	$+$	$\times$	Σ
MATLAB	30.29ms	2.27s	5.70ms
C	6.5ms	1.20s	0.95ms
NT2	7.1ms	1.25s	1.26ms
Γ_M	4.3	1.816	4.52
ε_C	91.5%	96%	75.4%

TAB. 3 – Temps d'exécution des fonctions de bases NT2

leur équivalent MATLAB. Pour les opérations purement algébriques, nous obtenons des performances équivalentes cette fois à celle de MATLAB, de par l'utilisation des fonctions de calcul LAPACK, le gain résiduel provenant des copies évitées par notre système d'évaluation. Ces résultats nous permettent donc de valider notre approche.

4.2 Traitement d'images

Après avoir validé les opérations élémentaires, nous nous proposons d'étudier comment NT2 se comporte au sein d'un algorithme plus complexe. Nous avons ici choisi d'implanter un algorithme de conversion de couleurs de l'espace RGB vers l'espace YUV. L'algorithme de conversion utilisé est le suivant : soit $P_{rgb} = (R,G,B)$, un pixel encodé sous le format RGB et $P_{yuv} = (Y,U,V)$ sa conversion au format YUV. On a alors :

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

On utilisera ici la représentation en virgule fixe de cette formule afin de ne pas avoir à effectuer de calcul en virgule flottante. Le code MATLAB correspondant est donnée dans le listing 3 et utilise une telle formulation en virgule fixe.

Listing 3 – Conversion RGB/YUV — Version MATLAB

```
function [Y,U,V]=rgb2yuv(I)
  R=I(:,:,1);
  G=I(:,:,2);
  B=I(:,:,3);

  Y=min(bitshift(abs(2104*R+4130*G+
                     802*B+135168),-13),235);
  U=min(bitshift(abs(-1214*R-2384*G+
                     3598*B+1052672),-13),240);
  V=min(bitshift(abs(3598*R-3013*G-
                     585*B+1052672),-13),240);
end
```

Son portage en C++ via NT2 tel qu'illustré dans le listing 4 est extrêmement simple, la quasi-totalité des fonctions et opérateurs utilisés étant identiques [6]. On notera l'utilisation du conteneur `view` pour effectuer l'extraction des canaux rouge, vert et bleu de l'image de départ. On peut alors le comparer à un code C équivalent, largement plus complexe (listing 5).

Les temps d'exécution, pour des images 128×128 à 1024×1024, de cet algorithme implanté sous MATLAB ,

[6]A l'exception de `bitshift`.

Listing 4 – Conversion RGB/YUV — Version C++

```
void RGB2YUV(const array<int>& I, array<int>& y,
                  array<int>& u, array<int>& v)
{
  view<int> R = I.ima_view(0);
  view<int> G = I.ima_view(1);
  view<int> B = I.ima_view(2);

  y=min(shr(abs(2104*R+4130*G+802*B+135168),13),235);
  u=min(shr(abs(-1214*R-2384*G+3598*B+1052672),13),240)
      ;
  v=min(shr(abs(3598*R-3013*G-585*B+1052672),13),240);
}
```

Listing 5 – Conversion RGB/YUV — Version C

```
void RGB2YUV(int* I, int* y, int* u, int* v, int sz)
{
  int  i;
  int* R = I;
  int* G = I+sz;
  int* B = I+2*sz;

  y = malloc( sizeof(int)*sz );
  u = malloc( sizeof(int)*sz );
  v = malloc( sizeof(int)*sz );

  for(i=0;i<sz;i++)
  {
    y[i]=min((abs(2104*R[i]+4130*G[i]+
              802*B[i]+135168) >> 13),235);
    u[i]=min((abs(-1214*R[i]-2384*G[i]+
              3598*B[i]+1052672) >> 13),240);
    v[i]=min((abs(3598*R[i]-3013*G[i]-
              585*B[i]+1052672) >> 13),240);
  }
}
```

en C et en C++ via NT2 sont donnés dans le tableau 4. On donne ensuite le gain obtenu en C++ par rapport à l'implantation MATLAB (Γ_M) et l'efficacité par rapport à une implantation directe en C (ε_C).

N	128	256	512	1024
MATLAB	44,6ms	175.8ms	487.5ms	1521.2ms
C	0.4ms	2.0ms	11.4ms	40.5ms
NT2	0.5ms	2.1ms	11.6ms	40.8ms
Γ_M	89.6	83.7	42.0	37.3
ε_C	80%	95.2%	98.2%	99.3%

TAB. 4 – Temps d'exécution de `RGB2YUV`

Les résultats sont très satisfaisants car le gain obtenu par NT2 est très élevé (de l'ordre de 50 à 100) et que cette vitesse d'exécution est comparable à celle obtenue par l'implantation directe en C.

4.3 Autres résultats

De la même manière, nous avons utilisé NT2 pour implanter divers algorithmes classiques de la vision artificielle comme un détecteur de point d'intérêts de Harris et Stephen [9], un détecteur de contour, un *trackeur* plan utilisant l'algorithme de Jurie/Dhome ou un algorithme de suivi par filtrage particulaire [6]. L'ensemble de ces implantations ont montré que NT2 permettaient d'obtenir des temps d'exécution comprise entre 75 et 90% du même algorithme implanté en C et gains de l'ordre de 25 à 100 par rapport à une implantation MATLAB, confirmant l'intérêt et les performances de NT2.

5 Conclusion

La définition d'outils performants pour la programmation d'application de vision artificielle n'est pas une tâche aisée. L'analyse des différents outils existants montre qu'il est difficile de fournir une implantation à la fois performante et de haut niveau et spécialement dans le cas du développement de bibliothèques dédiées. Le développement de ce type d'outil se heurte en général aux limitations du langage cible et ne peut garantir les performances finales des applications. Nous avons néanmoins montré qu'un tel développement est possible en utilisant un aspect relativement peu exploité dans le cadre du calcul scientifique : la méta-programmation *template.*

NT2 tire pleinement partie de ces techniques et permet donc d'utiliser un modèle de programmation simple et familier aux développeurs de la communauté Vision. Ses principaux apports au vu des modèles et outils de développement existants sont :

– Un modèle de programmation basé sur la manipulation de tableaux numériques à plusieurs dimensions grâce auquel la transition entre les modèles mathématiques et algorithmiques communément employés dans la communauté Vision et leur expression en tant que programme C++ est facilitée. NT2 propose en effet une interface très proche de celle de MATLAB, ce qui permet donc de réutiliser l'ensemble des solutions déjà développées grâce à cet outil de manière très rapide.

– Une implantation efficace qui permet de s'abstraire des limitations des implantations orientées objet classiques en C++. Cette implantation permet d'obtenir une fraction élevée des performances attendue d'une implantation directe en C et fournit un large panel d'options d'optimisations annexes.

La pertinence de NT2 a été validée à la fois sur des noyaux applicatifs simples mais aussi au sein d'applications plus complexes comme une application de reconstruction 3D temps réel [9] et une application de suivi de piéton temps réel [6].

Les perspectives de NT2 sont multiples et comportent par exemple une extension à des domaines connexes de la vision comme la commande ou le traitement d'image bas niveau. Les problématiques d'optimisations spécifiques à l'image – comme le choix des stratégies de gestions des bords d'une image ou du parcours du voisinage d'un point – sont actuellement à l'étude. A plus longue échéance, le

support des architectures multiprocesseurs ou multicœurs est envisagé. Ce support permettrait, via un marquage spécifique, de profiter de l'accélération de ces processeurs de manière transparente. Tout comme le support des extensions multimédia comme SSE2 ou AltiVec, le support des architectures multi-cœurs permettrait de simplifier la tâche du développeur afin de lui permettre de profiter des performances de ces machines qui, à court terme, seront de plus en plus répandues. Cette intégration nécessite néanmoins de revoir la structure même des méta-programmes internes fin de pouvoir extraire plus d'informations du contenu des expressions évaluées et de proposer des règles de sémantiques formelles idoines.

Références

[1] A. Alexandrescu. *Modern C++ Design : Generic Programming and Design Patterns Applied*. AW C++ in Depth Series. Addison Wesley, January 2001.

[2] J. Barton and L. Nackman. *Scientific and Engineering C++ : An Introduction with Advanced Techniques and Examples.* Addison-Wesley Professional, 1994.

[3] S. Chiba. A metaobject protocol for c++. In *OOPSLA '95 : Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, New York, NY, USA, 1995. ACM Press.

[4] K. Czarnecki. Generative programming : Methods, techniques, and applications. In *ICSR-7 : Proceedings of the 7th International Conference on Software Reuse*, pages 351–352, London, UK, 2002. Springer-Verlag.

[5] D. R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of USENIX Conference on Domain-Specific Languages*, pages 103–118, 1997.

[6] J. Falcou, T. Chateau, J. Sérot, and J. Lapresté. Real time parallel implementation of a particle filter based visual tracking. In *CIMCV 2006 - Workshop on Computation Intensive Methods for Computer Vision at ECCV 2006*, Grazz, 2006.

[7] J. Falcou and J. T. Lapresté. NT2 : The Numerical Template Toolbox, 2005. http ://nt2.sourceforge.net.

[8] J. Falcou and J. Sérot. E.V.E., An Object Oriented SIMD Library. *Scalable Computing : Practice and Experience*, 6(4) :31–41, December 2005.

[9] J. Falcou, J. Sérot, T. Chateau, and F. Jurie. A parallel implementation of a 3d reconstruction algorithm for real-time vision. In *PARCO 2005 - ParCo,Parallel Computing*, Malaga, Spain, September 2005.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[11] C. Harris and M. Stephens. A combined corner and edge detector. In *4th ALVEY Vision Conference*, pages 147–151, 1988.

[12] Intel Corporation. The OpenCV Vision Library. http ://www.intel.com/technology/computing/opencv/index.htm.

[13] Y. Ishikawa, A. Hori, M. Sato, and M. Matsuda. Design and implementation of metalevel architecture in c++ - mpc++ approach. In *Proceedings of the Reflection '96 Conference.*, pages 153–166, San Francisco, USA, 1996.

[14] C. Koebel, D. Loveman, R. Schreiber, G. S. Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, 1994.

[15] U. Köthe. Reusable software in computer vision. In P. G. B. Jähne, H. Haußecker, editor, *Handbook on Computer Vision and Applications*, volume 3. Academic PRess, 1999.

[16] K. Levenberg. A method for the solution of certain problems in least squares. *Quart. Appl. Math.*, 2 :164–168, 1944.

[17] D. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *SIAM J. Appl. Math.*, 11 :431–441, 1963.

[18] C. B. Moler. MATLAB — an interactive matrix laboratory. Technical Report 369, University of New Mexico. Dept. of Computer Science, 1980.

[19] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *ECCV*, pages 430–443, 2006.

[20] J. Sethian. Fast marching methods. *SIAM Review*, 41(2) :199–235, 1999.

[21] T. L. Veldhuizen. C++ templates are turing complete.

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

[23] T. L. Veldhuizen. C++ templates as partial evaluation. In O. Danvy, editor, *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18, San Antonio, January 1999. University of Aarhus, Dept. of Computer Science.

[24] T. L. Veldhuizen and D. Gannon. Active libraries : Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing (OO'98)*, Philadelphia, PA, USA, 1998. SIAM.

[25] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran ? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.