Compas'2016 : Parallélisme / Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Méthode de calcul de variance locale adaptée aux processeurs graphiques

Florian Gouin[1,2], Corinne Ancourt[1], Christophe Guettier[2]

MINES ParisTech/CRI - PSL Research University, Fontainebleau, France [1]
SAFRAN/Sagem Défense Sécurité, Massy, France [2]

Résumé
Le calcul de variance est couramment utilisé dans de nombreux domaines, par exemple en traitement d'images pour améliorer les contrastes locaux. Cet article présente un algorithme de calcul de variance pour processeur graphique en détaillant son optimisation en terme de précision et de temps de calcul vis-à-vis des contraintes architecturales liées au fonctionnement de type SIMD des GPU. Notre algorithme présenté dans cet article permet de réduire la complexité à $O(N \log N)$ et offre une accélération de 112 par rapport à la formulation usuelle et de 4 par rapport à l'algorithme *Pairwise*.

Mots-clés : Variance locale, GPU, granularité, traitement d'images

1. Importance du calcul de variance

Le calcul de variance est couramment employé dans de nombreux domaines. Simon et Litt [13] l'emploient par exemple dans le domaine de l'aérospatial. Le calcul de variance est alors utilisé pour surveiller en temps-réel les paramètres moteurs d'aéronefs dans le but d'améliorer la maintenance des moteurs par détection d'anomalies. Restrepo et al. [12] pour leur part, l'utilisent pour de la reconnaissance d'objet par une analyse volumétrique en 3 dimensions et Stünckler et Behnke [16] s'intéressent au calcul de variance appliqué à un algorithme de cartographie et de localisation simultanée.

L'ensemble de ces exemples n'est qu'un échantillon de cas d'utilisation et de manière plus générale, le calcul de variance est couramment utilisé dans le cadre de l'ANOVA [1], fondamentale aux domaines de l'analyse et de l'exploration de données.

Plus spécifiquement, en traitement d'images, Singh et al. [14, 15], Chang et Wu [3] ou encore Cvetkovic et al. [7, 8, 6, 5] utilisent la variance pour améliorer les contrastes locaux. Cette technique permet d'obtenir des images très contrastées, les zones sombres et les zones lumineuses étant désaturées. En revanche, ce type d'algorithme nécessite de maîtriser un certain nombre de problématiques comme l'ajout de bruit dans l'image ou encore la création d'artefacts en forme d'anneaux liés à l'exploitation de la variance. Ce dernier phénomène est visible dans la figure 1 ci-contre.

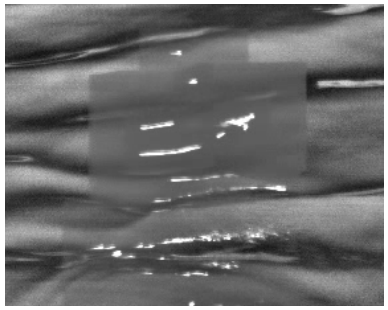

FIGURE 1 – Exemple de phénomène d'halos

1. Analyse de variance. De l'anglais **an**alysis **o**f **va**riance.

Cet article présente l'étude d'un algorithme de calcul de variance, son placement sur GPU et son application dans le cadre de l'amélioration de contrastes locaux en traitement d'images. Nous prenons en considération les solutions citées précédemment pour la gestion du bruit et des effets d'anneaux lors de l'application de la variance et apportons en plus une attention particulière au calcul amont de la variance locale. Nous traitons pour cela les problèmes de précision des calculs en virgule flottante, sources d'artefacts visuels importants, ainsi que les problèmes de quantité de communications et d'opérations arithmétiques détermistes pour les temps de calculs sur processeur graphique.
Benett et al. [1] se sont aussi intéressés au calcul parallélisé de variance. Cependant, leur étude s'appuie sur des clusters de calculs équipés de processeurs Intel Xeon. Leur architecture est donc différente de celle des GPU.
La solution que nous présentons dans cet article permet de réduire la complexité courante en $O(N^2)$ d'un tel algorithme à une complexité en $O(N \log(N))$ pour les communications mémoires ainsi que pour les opérations arithmétiques.

2. Contexte – Amélioration des contrastes locaux en traitement d'images

Le calcul de variance est appliqué dans le cadre de cet article à un algorithme de traitement d'images améliorant les contrastes locaux. Notre objectif est d'effectuer l'intégralité des calculs en temps-réel pour une séquence vidéo haute résolution de 1920 par 1080 pixels à 25 images par secondes ce qui laisse 40ms de temps de calcul disponible par image. Nous prenons en compte les solutions apportées par Singh et al. [14, 15], Chang et Wu [3] et Cvetkovic et al. [7, 8, 6, 5] afin de réduire le bruit et les effets d'artefacts en forme de halos. Ce point implique notamment d'effectuer nos calculs avec plusieurs tailles de noyaux de variance simultanément, ce qui augmente la quantité de calculs et de communications nécessaires.

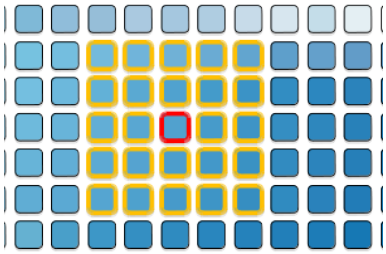

FIGURE 2 – Noyau de rayon 2 pixels

Le calcul de variance locale nécessite pour chaque élément constituant une image d'entrée, de calculer la variance locale comprise dans un noyau de taille donnée. La figure 2 donne un aperçu d'un noyau de 2 pixels de rayon composé des pixels entourés d'orange et de celui entouré de rouge. Ce dernier a la particularité d'être d'une part le centre du noyau et d'autre part l'élément de l'image pour lequel s'applique le noyau. Enfin, le même noyau de calcul est appliqué à chaque élément de l'image traitée. Nous avons de ce fait autant de données en entrée et en sortie d'algorithme. La gestion des bords se fera par réplication en miroir des données et sera confiée aux unités spécialisées du GPU.

3. Algorithmes de calcul de variance

Pébay [11] et Chan et al. [2] se sont intéressés à différents algorithmes de calcul de variance ainsi qu'à leur stabilité dans le cadre du calcul numérique. Les formules de calcul de variance citées dans la suite de cet article sont utilisées pour chaque élément d'une image afin de calculer leur variance locale.

Méthode 1 : formule de variance usuelle

$$\sigma_\varphi^2 = \frac{\sum_{i=1}^{n} (\varphi_i - \mu_{\varphi,n})^2}{n} \quad (1) \qquad \mu_{\varphi,n} = \frac{\sum_{i=1}^{n} \varphi_i}{n} \quad (2)$$

Pour un noyau de variance donné, n représente le nombre d'éléments de ce noyau, φ_i représente l'élément numéro i, $\mu_{\varphi,n}$ représente la moyenne de l'ensemble de taille n et σ_φ^2 représente le carré de l'écart-type, soit la variance de l'ensemble.

Dans cette méthode, le calcul de la moyenne locale (2) est invariant dans le calcul de la somme des écarts à la moyenne locale (1). De ce fait, il est plus judicieux d'extraire la boucle imbriquée calculant l'équation (2), de la boucle calculant la somme des écarts à la moyenne locale de l'équation (1). Cependant la dépendance générée par cette optimisation en nombre d'opérations et de communications impose de calculer ces 2 boucles successivement. Les éléments du noyau sont alors parcourus deux fois ce qui a pour effet de doubler la quantité d'accès mémoire. Enfin, la distributivité de ces deux boucles est faible à cause de la dépendance embarquée typique des boucles présentant une réduction. Les fonctions de coût en nombre de communications et en nombre d'opérations issues de notre implémentation, sont formulées par les équations (3) et (4).

$$\text{Coût}_{\text{Comm}} = \text{Img}_{size} \times (2N + 1) \quad (3) \qquad \text{Coût}_{\text{Op}} = \text{Img}_{size} \times (4N + 2) \quad (4)$$

Méthode 2 : formule de Kœnig

La formule de Kœnig (5) présente à l'inverse de la méthode précédente deux boucles indépendantes (2) et (6). Celles-ci ont le même nombre n d'itérations et traitent les mêmes données dans le même ordre φ_i. Il est de ce fait possible de fusionner ces deux boucles et ainsi de réduire la quantité d'accès mémoire, le parcours des données ne se faisant plus qu'une fois. Le résultat est visible en comparant les fonctions de coût (3) et (7).

$$\sigma^2_\varphi = \mu_{\varphi^2,n} - \mu^2_{\varphi,n} \quad (5) \qquad \mu_{\varphi^2,n} = \frac{\sum_{i=1}^{n} \varphi_i^2}{n} \quad (6)$$

Le comportement numérique de cette formule a cependant été démontré comme étant plus instable par Chan et al. [2] que la première méthode. En effet, plus la taille du kernel augmente plus les valeurs de μ_{φ^2} et μ^2_φ sont à la fois proches et élevées. En conséquence, la soustraction de ces deux valeurs génère une valeur faible et sujette à de fortes troncatures de la part des unités de calculs flottants du GPU comme l'expliquent Kirk et al. [10] et Collange et al. [4]. Enfin la technique du tri des données par ordre croissant pour améliorer la précision des calculs flottants citée par Whithead et al. [17] n'est pas envisageable du fait de l'augmentation des quantités d'opérations et de communications mémoires engendrées. La fonction de coût en nombre d'opérations est donnée par l'équation (8).

$$\text{Coût}_{\text{Comm}} = \text{Img}_{size} \times (N + 1) \quad (7) \qquad \text{Coût}_{\text{Op}} = \text{Img}_{size} \times (3N + 4) \quad (8)$$

Méthode 3 : algorithme *online*

$$M_{2,n} = M_{2,n-1} + (\varphi_n - \mu_{\varphi,n-1}) \times (\varphi_n - \mu_{\varphi,n}) \quad (9) \qquad \sigma^2_\varphi = \frac{M_{2,n}}{n} \quad (10)$$

L'algorithme *online* introduit une étape intermédiaire au calcul de variance avec $M_{2,n}$, détaillée dans la formule (9). Cette entité présente l'intérêt d'être numériquement plus stable que la méthode de Kœnig tout en conservant l'avantage de parcourir une seule fois les données du noyau. En revanche la dépendance entre $M_{2,n}$ et $M_{2,n-1}$ rend la parallélisation de cette méthode plutôt faible de part le traitement unitaire et séquentiel des données. Son comportement récursif présente de ce fait peu d'intérêt pour l'architecture de type SIMD des processeurs graphiques. Le calcul final de la variance est obtenu en utilisant l'équation (10). Les fonctions de coût en nombre de communications mémoires et d'opérations sont définies dans les équations (11) et (12).

$$\text{Coût}_{\text{Comm}} = \text{Img}_{size} \times (N + 1) \quad (11) \qquad \text{Coût}_{\text{Op}} = \text{Img}_{size} \times (6N + 2) \quad (12)$$

Méthode 4 : algorithme *Pairwise*

$$M_{2,\varphi_{1,2n}} = M_{2,\varphi_{1,n}} + M_{2,\varphi_{n+1,2n}} + \frac{1}{2n}\left(\sum_{i=1}^{n}\varphi_i - \sum_{i=n+1}^{2n}\varphi_i\right)^2 \quad (13)$$

L'algorithme *Pairwise* proposé par Chan et al. [2] est une généralisation de l'algorithme *online*. La formule (13) appliquée dans le cadre de cet algorithme permet de calculer la variance totale de deux sous-ensembles de taille identique dont la variance et la moyenne ont été calculées distinctement. L'algorithme propose une stabilité numérique similaire à celle de l'algorithme *online* et une parallélisation améliorée du fait que les sous-ensembles $M_{2,\varphi_{1,n}}$ et $M_{2,\varphi_{n+1,2n}}$ ne présentent pas de dépendance. Cette méthode, telle que décrite par Chan et al., est cependant plus adaptée à du calcul global de variance du fait qu'il s'agit d'une réduction progressive des données par itérations successives. Lors de chacune de ces itérations, le nombre de données est divisé par deux. Or la quantité de données en entrée et en sortie est identique pour un calcul de variances locales. Ainsi l'utilisation exacte de l'algorithme *Pairwise* en variance locale nécessite pour chaque noyau de variance de dupliquer les éléments concernés de l'image afin de les réduire. Cette action sacrifie ainsi le taux de communication au prix d'une distributivité améliorée. Les fonctions de coût en nombre d'opérations et de communications mémoires sont définies dans les équations (14) et (15).

$$\text{Coût}_{\text{Comm}} = \text{Img}_{size} \times (N-1) \times 6 \quad (14) \qquad \text{Coût}_{\text{Op}} = \text{Img}_{size} \times (N-1) \times 8 \quad (15)$$

4. Calcul de variance optimisé pour processeurs graphiques

4.1. Décomposition du noyau

La figure 3 ci-contre montre que pour deux éléments contigus d'une image, la quantité d'éléments communs au calcul des deux noyaux de variance impliqués croît de façon logarithmique pour tendre vers 100% au fur et à mesure que la taille du rayon du noyau augmente. Ces communications redondantes ainsi que celles présentes dans un voisinage plus large au fur et à mesure que la taille du noyau augmente, impactent les performances des algorithmes de calcul de noyaux en traitement d'image. Or, comme le font remarquer Hennessy et Patterson dans leur ouvrage de référence *Computer Architecture A Quantitative Approach* [9], la quantité de communications mémoires a un impact plus fort sur le temps d'exécution de GPU que la quantité d'opérations exécutées par les unités de calculs pour ce type d'algorithme. De ce fait, il est donc primordial de limiter ces communications mémoires redondantes. C'est dans ce but que nous proposons les deux axes suivants d'optimisation. Tous deux prennent en considération les problématiques d'artefacts en anneaux en utilisant une prépondérance centrale. Les éléments du noyau sont ainsi affectés d'un coefficient moins fort dans le calcul de la moyenne et de la variance pondérées au fur et à mesure que ceux-ci s'éloignent du centre du noyau.

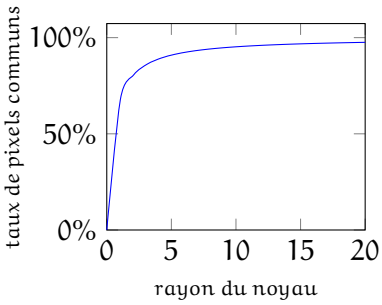

FIGURE 3 – Quantité de pixels communs entre deux kernels contigus en fonction de la taille du kernel

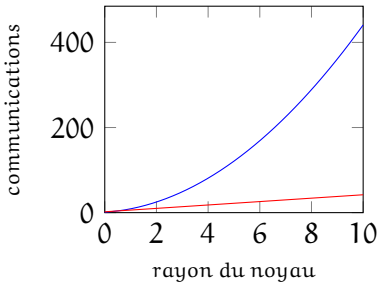

FIGURE 4 – Apport de la séparation de noyau sur la quantité de communications mémoire en fonction de la taille du kernel

La séparation de noyau

Afin de réduire les communications redondantes mises en évidence dans le précédent paragraphe, nous proposons d'appliquer une déconvolution à partir des deux dimensions indépendantes de notre noyau de variance pyramidale. Cette transformation illustrée par l'exemple (16),

permet de réduire notre quantité de communications en séparant le calcul d'un noyau de rayon r en deux vecteurs de taille $2r+1$ chacun. Dans notre exemple, r a pour valeur 1. Ainsi, nous réduisons notre complexité de coût initial en $O(N^2)$ avec $(2r+1)^2$ communications correspondant à la courbe bleue dans la figure 4 en une complexité en $O(N)$ avec $2 \times (2r+1)$ communications correspondant à la courbe rouge. Il est nécessaire d'appliquer successivement ces deux vecteurs. En revanche, l'équation (16) est commutative. Ainsi l'ordre d'application des deux vecteurs n'a pas d'importance.

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & 1 \end{pmatrix} \tag{16}$$

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 3 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \\ 2 \\ 0 \\ 1 \end{pmatrix} \tag{17}$$

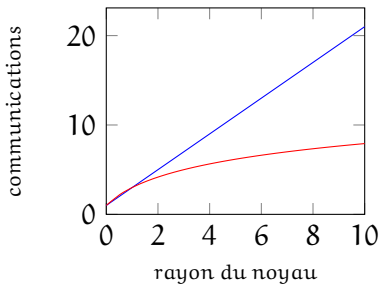

FIGURE 5 – Apport de la décomposition de vecteur sur la quantité de communications mémoire en fonction de la taille du kernel

La décomposition de vecteur

Afin de réduire encore la quantité de communications mémoires, nous proposons de transformer les deux vecteurs de dimensions indépendantes définis dans le paragraphe précédent en une série de vecteurs creux. L'exemple (17) illustre ce principe pour un vecteur de taille $2r+1$ avec $r=3$. L'ensemble de ces vecteurs a pour point commun d'avoir pour unique valeur non nulle, le centre du vecteur affecté d'un coefficient 2 et ses deux extrémités avec un coefficient 1. En procédant de la sorte, nous décomposons le calcul d'un vecteur en une série d'étapes successives et commutatives, faisant intervenir un vecteur creux de taille grandissante afin de faire évoluer la taille de notre vecteur global. Appliquée à l'ensemble des éléments d'une image, l'approche itérative de cette méthode permet de partager les résultats des calculs intermédiaires entre les éléments concernés. Cette optimisation contribue ainsi à réduire la quantité de communications et d'opérations comme le montre la figure 5 ci-contre. Ainsi, nous réduisons le coût de complexité en $O(N)$ de la séparation de noyau avec $2r+1$ communications représentées en bleu dans la figure ci-contre en une complexité en $O(\log(N))$ avec $2 \times \log_2(r+1)+1$ communications correspondant à la courbe rouge.

4.2. Notre algorithme Threewise

En modifiant l'algorithme *Pairwise* et en appliquant les deux optimisations listées précédemment, nous avons mis au point la formule (18) qui adresse plus précisément l'architecture spécifique des GPU pour effectuer le calcul de variance locale. Cette formule est adaptée aux deux optimisations citées précédemment. Ainsi, nous employons la séparation de noyaux en

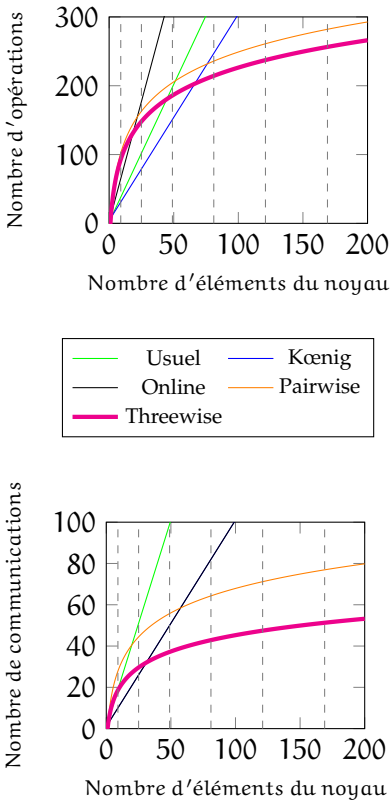

FIGURE 6 – Comparaison des fonctions de coût en communications mémoires et en opérations arithmétiques

deux vecteurs, eux-même décomposables en une série de vecteurs creux dont seuls 3 éléments sont non nuls. Ces trois éléments calculés pour chaque vecteur creux, sont en adéquation avec notre formule de variance adaptée au calcul de trois sous-échantillons dont l'un d'entre-eux est affecté d'un poids deux fois plus fort. Les fonctions de coûts en communications mémoires (19) et en opérations de calculs (20) résultant de l'implémentation de la formule (18) dans l'algorithme 1 sont représentées en mauve dans la figure 6. Les coûts de la formule usuelle sont représentés en vert, ceux de la formule de Kœnig en bleu et l'algorithme online en noir. Enfin, nous avons appliqué les optimisations de séparation de noyau et de décomposition de vecteur à l'algorithme *Pairwise* représenté par les courbes orange afin de réduire la complexité en $O(N \log N)$ et ainsi mieux évaluer le gain apporté par notre formule *Threewise*. Celle-ci est optimale à partir d'un noyau de 4 pixels de rayon pour le nombre d'opérations et à partir d'un noyau de 3 pixels de rayon pour la quantité de communications mémoires. Dans ce dernier cas, notre algorithme présente un gain notable par rapport à l'algorithme *Pairwise*.

$$M_{2,\varphi_{1,3n}} = M_{2,\varphi_{1,n}} + 2M_{2,\varphi_{n+1,2n}} + M_{2,\varphi_{2n+1,3n}} + \frac{\delta}{2n}$$
$$\delta = \left(\sum_{i=1}^{n}\varphi_i - \sum_{i=n+1}^{2n}\varphi_i\right)^2 + \frac{1}{2}\left(\sum_{i=1}^{n}\varphi_i - \sum_{i=2n+1}^{3n}\varphi_i\right)^2 + \left(\sum_{i=n+1}^{2n}\varphi_i - \sum_{i=2n+1}^{3n}\varphi_i\right)^2 \quad (18)$$

$$\text{Coût}_{\text{Comm}} = \text{Img}_{\text{size}} \times 2 \times (\log_2(N+1) - 1) \times 4 \quad (19)$$

$$\text{Coût}_{\text{Op}} = \text{Img}_{\text{size}} \times 2 \times (\log_2(N+1) - 1) \times 20 \quad (20)$$

5. Résultats expérimentaux

Afin de valider notre démarche théorique d'optimisation du temps d'exécution pour le calcul de variance locale, nous avons utilisé en entrée une image de format 1920 par 1080 pixels en niveaux de gris encodés sur 8 bits et un noyau de 63 pixels de rayon. Les algorithmes listés dans cet article ont été portés sur CUDA et le GPU utilisé est une carte NVIDIA Quadro K2000 avec une architecture Kepler. Le résultat du speedup des temps d'exécution est donné dans la figure 7 ci-contre. Notre algoritme présente un speedup de 112 pour 27 ms de temps de traitement par rapport à la formule usuelle qui nécessite un temps de 3,028 s. Enfin, comparé à la formule pairwise que nous avons améliorée et dont le temps de traitement est de 109 ms, notre algorithme présente un speedup de 4. L'algorithme de *Kœnig* et l'algorithme *online* ont des temps d'exécution respectifs de 2,221 s et 4,109 s. Nous noterons pour conclure, que l'algorithme *Threewise* a un temps d'exécution inférieur à 40 ms dans ce test ce qui permet de l'utiliser en temps-réel avec un flux vidéo à 25 Hz pour des images haute définition.

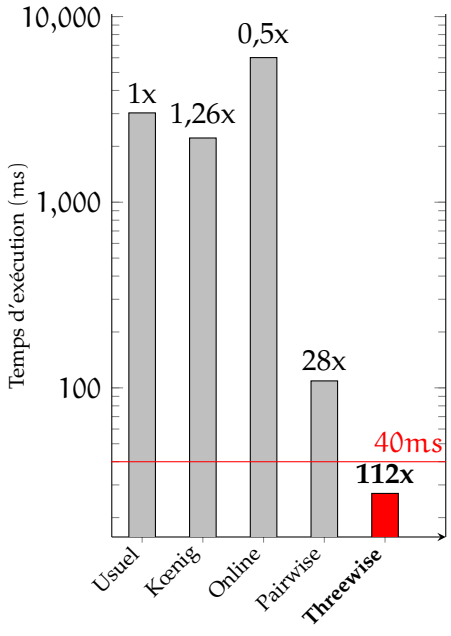

FIGURE 7 – Comparaison des temps d'exécutions et des facteurs d'accélérations

Input/Output: mImg une image de taille *W*IDTH × HEIGHT contenant l'image d'orgine
Input/Output: vImg une image de taille *W*IDTH × HEIGHT initialisée avec des 0

```
1  delta ← 1;
2  for s ← 0 to STEPS do
       /* Parcours du vecteur horizontal                                   */
3      for y ← 0 to HEIGHT do
4          for x ← 0 to WIDTH do
5              deltaA ← mImg[x][y] − mImg[x − delta][y];
6              deltaB ← mImg[x][y] − mImg[x + delta][y];
7              deltaC ← mImg[x + delta][y] − mImg[x − delta][y];
8              vImg2[x][y] ← vImg[x][y] + vImg[x − delta][y] + vImg[x + delta][y];
9              vImg2[x][y] ← vImg2[x][y] + (2 × deltaA × deltaA + 2 × deltaB × deltaB + deltaC × deltaC)/4;
10             mImg2[x][y] ← mImg[x][y] + mImg[x − delta][y] + mImg[x + delta][y];
       /* Parcours du vecteur vertical                                     */
11     Même boucles x et y mais delta est appliqué à y, on lit les données de mImg2 et vImg2 et on écrit dans mImg
       et vImg;
12     delta ← delta × 2;
```

Algorithm 1: Algorithme *Threewise* adapté au calcul de variance sur GPU

6. Conclusion

Nous avons présenté l'algorithme *Threewise* qui permet de calculer des noyaux de variance locale en traitement d'images. Celui-ci est dérivé de l'algorithme *Pairwise* et présente la même stabilité numérique. Grâce à deux optimisations, nous avons pu adapter ces deux algorithmes au calcul de noyau de variance sur GPU. Enfin, comparé aux autres algorithmes, notre algorithme *Threewise* réduit la quantité de communications mémoires et le nombre de calculs pour des noyaux de plus de 3 pixels de rayon et pour une complexité en $O(N \log N)$. Ces optimisations nous ont permis d'obtenir un speedup de 112 avec 27ms de temps de traitement dans notre expérimentation sur une NVIDIA Quadro K2000 par rapport à l'exploitation de la définition usuelle de la variance.

Bibliographie

1. Bennett, Janine and Grout, Ray and Pébay, Philippe and Roe, Diana and Thompson, David. – Numerically stable, single-pass, parallel statistics algorithms. – In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pp. 1–8. IEEE, 2009.
2. Chan, Tony F and Golub, Gene H and LeVeque, Randall J. – Updating formulae and a pairwise algorithm for computing sample variances. – In *COMPSTAT 1982 5th Symposium held at Toulouse 1982*, pp. 30–41. Springer, 1982.
3. Chang, Dah-Chung and Wu, Wen-Rong. – Image contrast enhancement based on a histogram transformation of local standard deviation. *Medical Imaging, IEEE Transactions on*, vol. 17, n4, 1998, pp. 518–531.
4. Collange, Sylvain and Daumas, Marc and Defour, David. – État de l'intégration de la virgule flottante dans les processeurs graphiques. *Technique et science informatiques*, vol. 27, n6, 2008, pp. 719–733.
5. Cvetkovic, S and Klijn, Jan and With, PHN de. – Tone-mapping functions and multiple-exposure techniques for high dynamic-range images. *Consumer Electronics, IEEE Transactions on*, vol. 54, n2, 2008, pp. 904–911.

6. Cvetkovic, S and Schirris, Johan and de With, PHN. – Non-linear locally-adaptive video contrast enhancement algorithm without artifacts. *Consumer Electronics, IEEE Transactions on*, vol. 54, n1, 2008, pp. 1–9.
7. Cvetkovic, Sascha D and Schirris, Johan and De With, Peter HN. – Locally-adaptive image contrast enhancement without noise and ringing artifacts. – In *Image Processing, 2007. ICIP 2007. IEEE International Conference on*volume 3, pp. III–557. IEEE, 2007.
8. Cvetkovic, Sascha D and Schirris, Johan and others. – Multi-band locally-adaptive contrast enhancement algorithm with built-in noise and artifact suppression mechanisms. – In *Electronic Imaging 2008*, pp. 68221C–68221C. International Society for Optics and Photonics, 2008.
9. Hennessy, John L and Patterson, David A. – *Computer architecture : a quantitative approach*. – Elsevier, 2011.
10. Kirk, David B and Wen-mei, W Hwu. – *Programming massively parallel processors : a hands-on approach*. – Newnes, 2012.
11. Pébay, Philippe. – Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. *Sandia Report SAND2008-6212, Sandia National Laboratories*, vol. 94, 2008.
12. Restrepo, Maria I and Mayer, Brandon A and Ulusoy, Ali O and Mundy, Joseph L. – Characterization of 3-d volumetric probabilistic scenes for object recognition. *Selected Topics in Signal Processing, IEEE Journal of*, vol. 6, n5, 2012, pp. 522–537.
13. Simon, Donald L and Litt, Jonathan S. – A Data Filter for Identifying Steady-State Operating Points in Engine Flight Data for Condition Monitoring Applications. *Journal of Engineering for Gas Turbines and Power*, vol. 133, n7, 2011, p. 071603.
14. Singh, S Somorjeet and Singh, Th Tangkeshwar and Devi, H Mamata and Sinam, Tejmani. – Local contrast enhancement using local standard deviation. *International Journal of Computer Applications*, vol. 47, n15, 2012.
15. Singh, S Somorjeet and Singh, Th Tangkeshwar and Singh, N Gourakishwar and Devi, H Mamata. – Global-Local Contrast Enhancement. *International Journal of Computer Applications*, vol. 54, n10, 2012.
16. Stückler, Jörg and Behnke, Sven. – Multi-resolution surfel maps for efficient dense 3D modeling and tracking. *Journal of Visual Communication and Image Representation*, vol. 25, n1, 2014, pp. 137–147.
17. Whitehead, Nathan and Fit-Florea, Alex. – Precision & performance : Floating point and IEEE 754 compliance for NVIDIA GPUs. *rn (A+ B)*, vol. 21, 2011, pp. 1–1874919424.