HAL
open science

Comparaison de la consommation énergétique et du temps d'exécution d'un algorithme de traitement d'images optimisé sur des architectures SIMD et GPU

Andrea Petreto, Arthur Hennequin, Thomas Koehler, Thomas Romera, Yohan Fargeix, Boris Gaillard, Manuel Bouyer, Quentin Meunier, Lionel Lacassagne

To cite this version:

Andrea Petreto, Arthur Hennequin, Thomas Koehler, Thomas Romera, Yohan Fargeix, et al.. Comparaison de la consommation énergétique et du temps d'exécution d'un algorithme de traitement d'images optimisé sur des architectures SIMD et GPU. Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS 2018), Jul 2018, Toulouse, France. hal-01835219

HAL Id: hal-01835219

https://hal.science/hal-01835219

Submitted on 11 Jul 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Compas'2018 : Parallélisme / Architecture / Système
ENSEEIHT - Toulouse, France, du 03 au 06 juillet 2018

Comparaison de la consommation énergétique et du temps d'exécution d'un algorithme de traitement d'images optimisé sur des architectures SIMD et GPU

Andrea Petreto[1,2], Arthur Hennequin[1], Thomas Koehler[1], Thomas Romera[1], Yohan Fargeix[1], Boris Gaillard[2], Manuel Bouyer[1], Quentin Meunier[1], Lionel Lacassagne[1]

Sorbonne Université, CNRS, LIP6 – Laboratoire d'Informatique de Paris 6 [1]
F-75005 Paris, France – prenom.nom@lip6.fr

Lhéritier - Alcen – F-95862, Cergy-Pontoise, France [2]
bgaillard@lheritier-alcen.com

Résumé
Cet article présente et compare les implémentations optimisées d'un algorithme de flot optique sur des cartes embarquées à base de processeurs SIMD multicoeurs et de GPU. La comparaison est effectuée à la fois en termes de vitesse de calcul – pour atteindre une cadence de traitement temps réel – et en termes d'énergie. Les résultats obtenus montrent que les GPU sont les plus efficaces à la fois en termes de vitesse et de consommation, pouvant traiter dans la meilleure configuration 25 images de 8M pixels par secondes pour 0.35 joule par image.

Mots-clés : flot optique, efficacité énergétique, optimisation de code, architectures multicoeur SIMD/GPU, exécution temps réel.

---

1. Introduction

Les systèmes embarqués doivent satisfaire des contraintes antagoniste de cadence de traitement et de consommation d'énergie. C'est le cas du projet nanosat Meteorix de Sorbonne Université [16] qui a pour but la détection de météores en temps réel depuis une orbite de 500 km. Des étudiants de différentes disciplines se sont investis dans ce travail, encadrés par des enseignants-chercheurs, des ingénieurs et des doctorants. La chaîne de traitement en cours de développement est basée sur une analyse de la vitesse apparente de tâches lumineuses pour discriminer villes, éclairs et météores. Les vidéos utilisées pour la validation qualitative de la chaîne de traitement (non abordée dans cet article) ont été acquises depuis l'intérieur de la station spatiale internationale ISS [19] par la caméra *Meteor* de l'Université de Chiba [5].
Cet article étudie et compare différentes implémentations d'un algorithme de flot optique – permettant de détecter les pixels en mouvement et servant de base à la détection – d'un point de vue vitesse et consommation énergétique, et met en évidence l'impact des optimisations algorithmiques et architecturales pour les processeurs SIMD et les GPU.
La section 2 présente l'algorithme de flot optique utilisé ; la section 3 présente l'ensemble des optimisations ; la section 4 présente le protocole de mesure ainsi que les plate-formes de tests ; la section 5 présente et analyse les résultats obtenus ; enfin, la section 6 conclut.

2. Algorithmes itératifs de flot optique

Les algorithmes de flot optique ont pour but d'estimer la vitesse apparente en tout point pour tout couple d'images. Une analyse qualitative particulièrement exhaustive [1] des algorithmes existants est disponible sur le site Middlebury [17], qui fournit aussi des liens vers des codes. Le site IPol [11] du CMLA fourni aussi plusieurs codes d'algorithmes récents. Leur implémentation optimisée a fait l'objet de nombreux travaux essentiellement sur FPGA [8, 2, 14, 4] et sur GPU [22, 12, 13, 24] mais peu sur CPU [13, 7]. On notera également que de nouvelles méthodes d'estimation du flot basées sur l'apprentissage gagnent en popularité auprès de la communauté [25, 23]
Pour le projet Meteorix, l'algorithme retenu est celui de Horn-Schunk [10]. Ce n'est pas le plus précis, mais il a les avantages d'être simple et d'avoir fait l'objet de très nombreuses publications. Sa structure de calcul est adaptée à des implémentations sur CPU et GPU, et de nombreux algorithmes plus évolués ont une structure similaire, pour lesquels il peut donc servir de référence. Son fonctionnement est le suivant : l'algorithme calcule initialement les dérivées spatio-temporelles (I_x, I_y I_t) du couple d'images en entrée, puis calcule itérativement une moyenne sur un voisinage local (eq.1), ainsi que la mise à jour en tout point du champ de vecteurs vitesse (u, v) (eq. 2). Le terme α^2 est un paramètre de convergence.

$$\bar{u} = \frac{1}{12}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 0 & 2 \\ 1 & 2 & 1 \end{bmatrix} * u \qquad \bar{v} = \frac{1}{12}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 0 & 2 \\ 1 & 2 & 1 \end{bmatrix} * v \tag{1}$$

$$u = \bar{u} - I_x \times \frac{I_x \times \bar{u} + I_y \times \bar{v} + I_t}{\alpha^2 + I_x^2 + I_y^2} \qquad v = \bar{v} - I_y \times \frac{I_x \times \bar{u} + I_y \times \bar{v} + I_t}{\alpha^2 + I_x^2 + I_y^2} \tag{2}$$

Afin de simplifier l'analyse de performance et de la rendre plus simple à reproduire, c'est la version mono-résolution de l'algorithme qui est évaluée, car elle permet d'avoir de bons estimateurs des temps de calcul et des mesures de consommation, sans se confronter à des problèmes d'ordre qualitatif (algorithme d'interpolation, nombre d'échelles) de la version pyramidale.

3. Optimisations pour la vitesse et la consommation

L'optimisation d'un système embarqué en vitesse [6] et en consommation [3, 9, 18] peut se faire en minimisant un des deux critères sous l'hypothèse que l'autre critère respecte une contrainte dure. L'approche proposée ici consiste plutôt à étudier la frontière efficiente des points de fonctionnement dans l'espace (vitesse, consommation) pour des codes à différents niveaux d'optimisation, sur différents processeurs, et à différentes fréquences de fonctionnement. Les compromis trouvés pourront par exemple déterminer la taille maximale d'image analysable selon la configuration, ainsi qu'alimenter la réflexion sur les besoins en énergie.
Les transformations algorithmiques considérées ici pour l'optimisation sont celles décrites dans [15], notamment le pipeline d'opérateurs et la fusion d'opérateurs. L'algorithme de Horn-Schunk se prête particulièrement bien à ce type de transformations : il est possible de fusionner les étapes de calcul de la moyenne et de mise à jour pour diminuer le nombre d'accès mémoire et améliorer l'intensité arithmétique. Mais plus important, l'aspect itératif est source de deux autres optimisations : la fusion des champs de vecteurs vitesse (source et destination ne sont qu'une seule zone mémoire) et le pipeline spatial et temporel. Ainsi, au lieu de réaliser une mise à jour complète de (U, V) sur la totalité des matrices, le traitement est pipeliné par ligne. Ces transformations permettent de maximiser la persistance des données à l'intérieur des processeurs (dans les caches pour les CPU, dans la mémoire partagée pour les GPU), tout en

minimisant les transferts avec la mémoire externe. Nous considérons dans la suite 8 itérations par pixel de cet algorithme.
Les formats de calcul peuvent aussi être optimisés en passant de flottants 32-bit (F_{32}) à des flottants 16-bit (F_{16}) sur GPU, ce qui permet plus de parallélisme de données et une meilleure bande passante, tout en assurant une précision suffisante des calculs [21]. Ce format n'est – à notre connaissance – toujours pas disponible sur CPU.

4. Évaluation expérimentale

Les différentes cartes utilisées sont présentées dans la table 1. Parmi les 5 cartes, 3 possèdent un GPU, pour un total de 8 architectures étudiées. Pour chaque architecture, différentes implémentations sont évaluées. Pour les architectures CPU, toutes les versions sont multi-threadées avec un thread par coeur fixé. Les variantes considérées sont l'algorithme de base (*base*), deux versions SIMD non pipelinées avec respectivement 1 ou 2 buffers (*simd 1* et *simd 2*), et deux versions SIMD pipelinées avec 1 ou 2 buffers (*pipe 1* et *pipe 2*). Pour les GPU sont considérées la version de base de l'algorithme (*base*) et une version optimisée (*opt*) comprenant les optimisations suivantes : mémoire partagée, regroupement de U et V, fusion des opérateurs, et itérations locales aux blocs. De plus, ces deux versions sont déclinées en F_{32} et en F_{16}.
Afin de réaliser une mesure de puissance à la fois simple et reproductible, nous mesurons la consommation de la carte complète. Pour cela nous avons développé une carte qui s'insère entre la source d'alimentation et la carte de calcul. La tension (à travers un pont diviseur), et le courant (par une résistance en série et un amplificateur de tension) sont mesurés par un micro-contrôleur. Ce dernier inclus également l'état de 4 GPIO venant de la carte de calcul, afin de faire correspondre les points de mesures à des événements venant du logiciel de calcul. Le micro-contrôleur envoie 5000 échantillons par seconde à un PC hôte. Nous utilisons une alimentation de laboratoire, et la carte de mesure a été calibrée.

TABLE 1 – Spécification des cartes évaluées

carte	techno	CPU	Fmax (GHz)	GPU	Fmax (MHz)
PCduino8	28 nm	8×A7	1.80	-	-
Raspberry Pi 3	40 nm	4×A53	1.20	-	-
Jetson TK1	28 nm	4×A15	2.32	192 C Kepler	852
Jetson TX1	20 nm	4×A57	1.73	256 C Maxwell	998
Jetson TX2	16 nm	4×A57 (+ 2×Denver2)	2.00	256 C Pascal	1300

Les mesures de performances et de consommation sont effectuées simultanément, en faisant varier la fréquence du processeur et la taille des images. Les fréquences sont prises parmi les fréquences disponibles sur chaque carte. Lorsque cela est possible, la fréquence de la mémoire externe est fixée à son maximum. Le système de refroidissement intégré de la carte est aussi activé au maximum (quand celui-ci existe) et les processus d'économie d'énergie de l'OS sont désactivés. Nous avons utilisé tous les coeurs ARM de chaque carte, mais désactivé les coeurs Denver2 de la Jetson TX2. Les images de test sont des images carrées, de côté variant entre 64 et 1024 pixels avec un pas de 8 pour les versions CPU et entre 208 et 2048 avec un pas de 16 pour les versions GPU. La taille de 1008 souvent utilisée correspond à la plus grande taille qui ne soit pas un multiple de la taille de cache et compatible avec le découpage des GPU.
Concernant les GPU, les performances sont mesurées pour chaque *kernel* [20] pour des blocs de 64 à 512 *threads*, en prenant des multiples de 32 (taille d'un *warp* [20]), 8 à 256 *threads* de largeur et 1 à 32 *threads* de hauteur. La meilleure taille de bloc est ensuite utilisée pour les comparaisons.

5. Résultats et Analyse

5.1. Points de fonctionnement et frontière efficiente

La figure 1 montre la frontière efficiente des points de fonctionnement dans l'espace (temps par pixel, consommation par pixel) pour chacune des configurations étudiées.

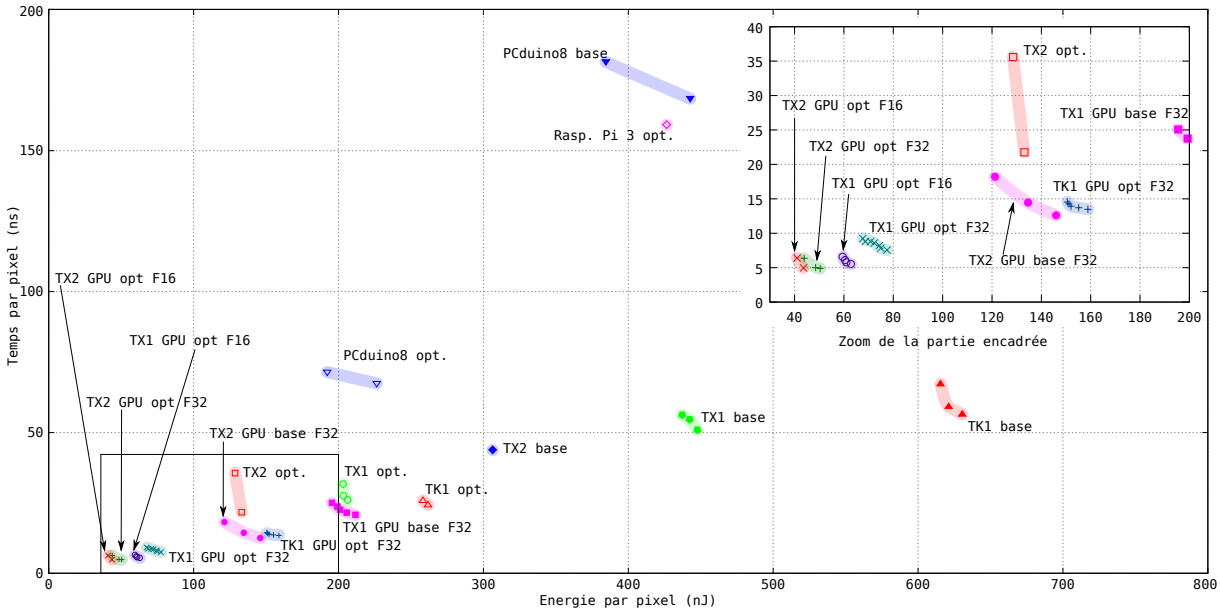
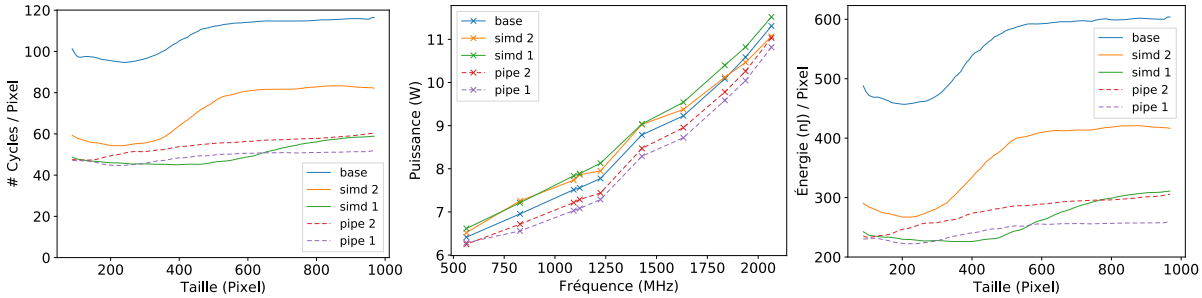


FIGURE 1 – Frontière efficiente des points de fonctionnements dans l'espace (temps/pixel, consommation/pixel), pour les architectures et algorithmes étudiés. Chaque point représente une fréquence donnée. La version *opt.* pour les processeurs généralistes est la version la plus rapide (*pipe 1*). La configuration Raspberry Pi 3 *base* se trouve en dehors de l'espace représenté (énergie = 1145 nJ, temps = 415 ns)

La meilleure configuration en GPU est obtenue avec la carte TX2, et il en est de même pour les versions CPU uniquement. Cette figure met en évidence la performance des algorithmes sur GPU, ainsi que l'impact des optimisations apportées. Pour mieux expliquer les phénomènes observés, nous avons effectué une série d'autres mesures, présentées dans les sections suivantes.

5.2. Impact des optimisations CPU

La figure figure 2(a) montre l'impact des différentes optimisations sur la vitesse de traitement sur la carte Jetson TK1 – les autres cartes testées exhibant un comportement similaire. En faisant varier la taille des images, nous observons une sortie de cache pour les versions *base*, *simd 2* et *simd 1*. La version *simd 1* permet de reporter la sortie de cache pour des images plus grande. Les versions *pipe 1* et *pipe 2* ne présentent pas de sortie de cache sur l'intervalle de tailles mesuré ce qui leur permet de rester rapide pour de plus grandes tailles d'images.
Nous observons sur la figure 2(b) que l'utilisation des unités SIMD augmente très peu la consommation par rapport à la version *base*. Cette figure montre aussi l'impact des accès à la mémoire externe sur la consommation globale : les versions pipelinées *pipe 1* et *pipe 2* permettent de réduire le nombre d'accès hors du cache et réduisent donc la puissance consommée. L'énergie consommée par pixel étant proportionnelle au temps de calcul et à la puissance, les versions pipelinées cumulent les avantages, comme on peut le voir sur la figure 2(c).

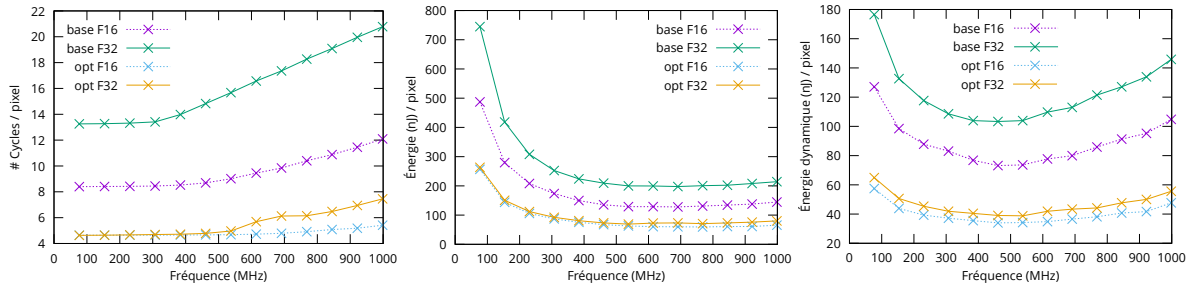


(a) CPP = f(Taille) (b) Puissance = f(Fréquence) (c) Énergie = f(Taille)

FIGURE 2 – Impact des optimisations sur la vitesse et la consommation des CPU (Jetson TK1 @2.0655GHz pour (a) et (c), images de taille 1008 pour (b))

5.3. Impact des optimisations GPU

La figure 3 montre l'évolution du nombre de cycles par pixel et de l'énergie en fonction de la fréquence pour la carte TX1, sur les codes de base et optimisés, en F_{16} et en F_{32}. La figure 3(a) montre qu'en F_{32}, la version optimisée est environ 3 fois plus rapide que la version de base. La version de base est entre 1.5 et 2 fois plus rapide en F_{16} qu'en F_{32}. Ce n'est pas le cas pour la version optimisée où le F_{16} ne devient plus rapide que dans les hautes fréquences. En effet, tant que la mémoire est suffisamment rapide par rapport au processeur, il n'y a pas de gain en F_{16} car les calculs ne sont pas plus rapides par manque d'ILP (*Instruction Level Parallelism*) dans l'implémentation. Lorsque la mémoire devient un facteur limitant, le F_{16} devient plus rapide car il demande 2 fois moins de bande passante que le F_{32}. C'est ce même phénomène qui explique la montée des cycles par pixel vers les hautes fréquences : le débit nécessaire pour maintenir la cadence devient supérieur à la bande passante maximum de la carte. Le code devient dans ce cas *memory bound*.



(a) Nombre de cycles / pixel (b) Énergie (c) Énergie dynamique

FIGURE 3 – Vitesse et consommation en fonction de la fréquence pour les versions du GPU TX1, images de taille 1008

La figure 3(b) montre que la tendance de consommation entre les différentes version est la même que pour la vitesse : la variation de puissance est négligeable face à la variation du temps de consommation. Sur la figure 3(c), sur laquelle est soustraite la consommation au repos, on remarque que les fréquences les plus efficaces énergétiquement correspondent à la transition *Compute Bound - Memory Bound* où les cycles par pixel commencent à se dégrader. Prendre en compte la consommation au repos augmente légèrement la fréquence énergétiquement optimale.

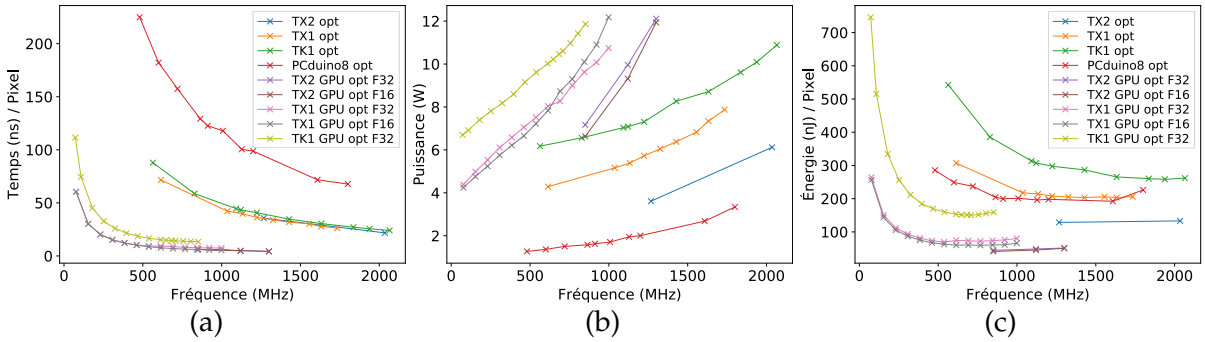


FIGURE 4 – Vitesse (a), puissance (b) et énergie (c) des différentes architectures en fonction de la fréquence. La légende de la figure centrale, omise par manque de place, est identique.

5.4. Comparaison des architectures et processeurs étudiés

La figure 4(a) montre que les GPU sont plus rapides que les CPU à la même fréquence (environ $\times 4$ sur la TX1 vers 1GHz). Ceci est cohérent car les GPU sont massivement parallèles. Les GPU restent plus rapide vers 1GHz que les CPU vers 2GHz. Le CPU de la TX1 apporte une accélération par rapport à celui de la TK1 mais reste proche. Sa fréquence maximale est moins élevée mais apporte la même vitesse. Le GPU de la TX1 apporte une bonne accélération ($\times 2$) par rapport à celui de la TK1. Sa fréquence maximale est aussi plus élevée.
La figure 4(b) montre que la consommation instantanée des différents processeurs varie quasi-linéairement en fonction de la fréquence, mais avec des pentes différentes. La consommation instantanée des GPU est plus importante que celles des CPU, à fréquences équivalentes, mais les GPU peuvent descendre plus bas en fréquence afin de réduire leur consommation instantanée. Nous avons également mesuré l'énergie totale dépensée par chacun des processeurs (figure 4(c)) afin de faire une comparaison plus juste. Cela permet de choisir une fréquence de fonctionnement optimale pour chacun des processeurs et de voir que les GPU évalués sont plus efficaces énergétiquement que les CPU.

5.5. Synthèse

Les meilleures configurations sur CPU et GPU sont obtenues sur la carte TX2. La carte PCduino a été ajoutée car c'est celle dissipant le moins de Watts. La table 2 résume ces configurations et donne la plus grande taille d'image pouvant être traitée à la cadence de 25 images par seconde.

TABLE 2 – Meilleures configurations obtenues, avec taille d'image équivalente à 25 fps.

configuration	E (nJ/pixel)	t (ns/pixel)	taille max (#pixels)
TX2 GPU opt F16 énergie min.	41.0	6.39	2501
TX2 GPU opt F16 temps min.	43.6	4.96	2839
TX2 CPU opt énergie min.	128.6	35.6	1059
TX2 CPU opt temps min.	133.2	21.8	1355
PCduino opt énergie min.	192.2	71.8	746
PCduino opt temp min.	226.4	67.7	768

6. Conclusion

Cet article présente une comparaison de plusieurs implémentations de l'algorithme de Horn-Schunk, servant à la détection de mouvement dans une image, sur différentes architectures SIMD et GPU, dans le but de réduire à la fois la consommation et le temps de traitement. Les configurations les plus efficaces permettent de traiter – à la cadence de 25 images/s – des

images carrées de taille 2839 pixels sur GPU et 1355 sur CPU. Parmi les travaux futurs, nous envisageons de regarder la précision des calculs au format virgule fixe 16 bits, afin de pouvoir doubler le parallélisme SIMD sur CPU. Enfin, nous visons une comparaison avec l'algorithme TV-L1 comprenant une analyse qualitative des images.

Remerciements

Ce travail a été en partie subventionné par une thèse DGA, l'ESEP et Janus CNES. L'équipe Meteorix tient à remercier Tomoko Arai du projet PERC de l'Université de Chiba pour la fourniture de séquences vidéo, ainsi que Jean-Michel Morel et son équipe du CMLA de l'ENS Cachan.

Bibliographie

1. Baker (S.), Lewis (D. S. J. P.), Roth (S.), Black (M. J.) et Szeliski (R.). – A database and evaluation methodology for optical flow. *International Journal of Computer Vision*, vol. 17,1, 2011, pp. 1–31.
2. Bako (L.), Hajdu (S.), Brassai (S.-T.), Morgan (F.) et Enachescu (C.). – Embedded implementation of a real-time motion estimation method in video sequences. *Procedia Technology*, vol. 22, 2016, pp. 897–904.
3. Basmadjian (R.) et de Meer (H.). – Evaluating and modeling power consumption of multi-core processors. – In *Proceedings of the 3rd International Conference on Future Energy Systems : Where Energy, Computing and Communication Meet*, p. 12. ACM, 2012.
4. Chai (Z.), Zhou (H.), Wang (Z.) et Wu (D.). – Using c to implement high-efficient computation of dense optical flow on FPGA-accelerated heterogeneous platforms. – In *International Conference on Field-Programmable Technology (FPT)*, pp. 260–263. IEEE, 2014.
5. Chiba. – Meteor project `http://www.perc.it-chiba.ac.jp/project/meteor/gallery.html`.
6. Datta (K.), Murphy (M.), Volkov (V.), Williams (S.), J.Carter, Oliker (L.), Patterson (D.), Shalf (J.) et K.Yelick. – Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. – In *Supercomputing*, pp. 1–12. ACM/IEEE, 2008.
7. Garcia-Dopico (A.), Pedraza (J. L.), Nieto (M.), Pérez (A.), Rodríguez (S.) et Navas (J.). – Parallelization of the optical flow computation in sequences from moving cameras. *EURASIP Journal on Image and Video Processing*, vol. 2014, n1, 2014, p. 18.
8. Gultekin (G.) et Saranli (A.). – An FPGA-based high performance optical flow hardware design for computer vision. *Microprocessors and Microsystems*, vol. 37, n1-3, 2013, pp. 270–286.
9. Hager (G.), Treibig (J.), Habich (J.) et Wellein (G.). – Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency and computation : practice and experience*, vol. 28, n2, 2016, pp. 189–210.
10. Horn (B. K. P.) et Schunk (B. G.). – Determining optical flow. *ACM Computing Surveys (CSUR)*, vol. 17, n1-3, 1981, pp. 185–203.
11. IPOL. – Image processing on line `http://www.ipol.im/`.
12. J.D.Adarve et Mahony (R.). – A filter formulation for computing real time optical flow. – In *(IEEE Robotic and Automation Letters)*, pp. 1192–1199. IEEE, 2016.
13. Kroeger (T.), Timofte (R.), Dai (D.) et Gool (L. V.). – Fast optical flow using dense inverse search. – In *(ECCV)*, 2016.
14. Kunz (M.), Ostrowski (A.) et Zipf (P.). – An FPGA-optimized architecture of horn and schunck optical flow algorithm for real-time applications. – In *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4. IEEE, 2014.
15. Lacassagne (L.), Etiemble (D.), Hassan-Zahraee (A.), Dominguez (A.) et Vezolle (P.). – High level transforms for SIMD and low-level computer vision algorithms. – In *ACM Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, pp. 49–56, 2014.
16. Meteorix. – Nanosat Sorbonne Université `http://www.nanosat.upmc.fr/fr/index.html`.
17. Middlebury. – Optical flow database `http://vision.middlebury.edu/flow/`.

18. Mittal (S.) et Vetter (J. S.). – A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, vol. 47, n4, 2015, pp. 1–36.
19. NASA. – Monitoring meteor showers from space `http://https://directory.eoportal.org/web/eoportal/satellite-missions/content/-/article/iss-meteor`.
20. NVIDIA. – Cuda, programming model `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model`.
21. Piskorski (S.), Lacassagne (L.), Bouaziz (S.) et Etiemble (D.). – Customizing CPU instructions for embedded vision systems. – In *Computer Architecture, Machine Perception and Sensors (CAMPS)*, pp. 59–64. IEEE, 2006.
22. Plyer (A.), Besnerais (G. L.) et Champagnat (F.). – Massively parallel lucas kanade optical flow for real-time video processing applications. *Journal of Real-Time Image Processing*, vol. 11,4, 2016, pp. 713–730.
23. Ranjan (A.) et Black (M. J.). – Optical flow estimation using a spatial pyramid network. – In *(CVPR)*, pp. 2720–2729. IEEE, 2017.
24. Senst (T.), Evangelio (R. H.), Keller (I.) et Sikora (T.). – Clustering motion for real-time optical flow based tracking. – In *Advanced Video and Signal-Based Surveillance (AVSS), 2012 IEEE Ninth International Conference on*, pp. 410–415. IEEE, 2012.
25. Weinzaepfel (P.), Revaud (J.), Harchaoui (Z.) et Schmid (C.). – Deepflow : Large displacement optical flow with deep matching. – In *International Conference on Computer Vision (ICCV)*, pp. 1385–1392. IEEE, 2013.