RENPAR'16 / CFSE'4 / SympAAA'2005 / JC
Le Croisic, France, 5 au 8 avril 2005

Intégration système d'une application de détection d'obstacles par stéréovision

Daniel Zaoui
Département ASIM du LIP6
Université Paris VI
Paris

Frédéric Pétrot*
Laboratoire TIMA
Institut National Polytechnique
Grenoble

Raphaël Labayrade
LIVIC
École des mines
Paris

Résumé
Cet article présente l'étude architecturale d'un capteur permettant d'effectuer la détection d'obstacles par stéréovision embarquée dans un véhicule. Deux caméras, placées à l'avant du véhicule, fournissent au capteur deux images de la même scène prises de points de vue différents. L'objectif est d'abstraire dans le temps imparti la route et les obstacles, afin de déduire l'espace roulable. Cette information est ensuite transmise à un ordinateur de bord qui prend des décisions sur le contrôle du véhicule. L'intégration de cette application exige une analyse fine des algorithmes afin de définir les contraintes d'implantation des différentes parties qui la constitue. En fonction des contraintes, une architecture logicielle/matérielle est choisie. Cette architecture est alors décrite au bit près en vue d'être simulée grâce à un simulateur précis au cycle, ce qui permet de valider ou d'invalider les choix d'implantation.

Mots-clés : architecture système, intégration, stéréovision, méthode de conception système

1. Introduction

L'objectif de ce travail est d'implanter sous forme de système sur puce une application de détermination de l'espace roulable, en détectant par vision stéréoscopique la route et les obstacles qui peuvent s'y trouver. L'application est fournie par le Laboratoire des Interactions Véhicule-Infrastructure-Conducteur (LIVIC)[6]. Elle est entièrement logicielle et opérationnelle sur des images 320×200 en niveau de gris codés sur 8 bits par pixel.
Pour être viable économiquement, l'application doit être intégrée dans une seule puce, avec potentiellement de la mémoire externe. Pour être capable de détecter les obstacles de manière extrêmement fiable, il faut traiter 100 images par secondes, les images étant à la fois plus grandes et avec plus de niveau de gris.
Ce papier propose donc une étude détaillée de l'application en vue de son intégration, puis un choix d'architecture et une implantation effective. Des simulations précises au cycle et au bit près permettent de valider fonctionnellement l'implantation et soit de s'assurer qu'elle respecte les contraintes, soit de constater qu'elle les viole.

2. Présentation de l'application

Deux caméras observent une scène sous deux angles différents, de sorte qu'un point de la scène n'a pas la même position sur les deux images. Cet écart de position est appelé disparité. Si on est

* Ce travail a été effectué lorsque F. Pétrot était au LIP6

capable d'attribuer une disparité à chaque point de l'image, on pourra aisément reconstruire la scène 3D (plus précisément la carte de profondeur). La récupération des contours, appelés *primitives*, se fait par un calcul de gradient sur tous les points de la paire stéréoscopique. L'application permettant la détection d'obstacles se trouvant sur la route, il faut tout d'abord déterminer la route proprement dite, ce qui se fait à partir de la carte de disparité. Il reste ensuite à déterminer les obstacles, à savoir les primitives ne constituant pas la route. Enfin, dès que l'on connaît la nature de chaque primitive (route ou obstacle), on peut en extraire l'espace roulable.
Les différentes étapes du traitement sont brièvement présentées maintenant, avec leur complexité. On notera I une image, l'indice d indiquant une image droite, l'indice g une image gauche. De plus, L est le nombre de lignes et N le nombre de pixels par ligne de I.

2.1. Extraction des primitives

Cette étape consiste en l'extraction des primitives de chaque image, à savoir les contours des objets grâce à un calcul de gradient. Si i représente l'indice dans la ligne du point sur lequel on travaille et $I(i)$ son niveau de gris, $I(i+1) - I(i)$ est le différentiel en ce point. Si cette valeur est supérieure à un seuil prédéfini, ce point est une primitive à gradient positif. Si elle est inférieure à l'opposé de ce seuil, c'est une primitive à gradient négatif. Cet algorithme est en $O(N)$ pour chaque ligne, donc en $O(N \times L)$ pour une image entière. En moyenne, pour des scènes réalistes, au maximum 10% des points sont conservés (5% par signe de gradient). Un exemple d'image provenant des caméras et l'extraction effectuée (primitives à gradient positif en blanc, primitives à gradient négatif en gris) est présenté figure 1.

FIG. 1 – Une image et le résultat d'une extraction de gradient

2.1.1. Filtrage du bruit

L'étape d'extraction des primitives nécessite un filtrage afin d'éliminer les points isolés. Il faut pour cela analyser l'ensemble des primitives extraites et déterminer pour chacune d'elles le nombre de voisins. Si celui-ci est suffisant, on conserve la primitive. Pour chaque ligne, on a en moyenne 5% de primitives. Le voisinage étudié est un carré de 5×5. Ainsi, 25 tests sont à effectuer par primitive. On obtient donc une complexité en $O(25 \times N \times 5\%)$ par ligne, qu'il faut multiplier par L pour l'image entière.

2.2. Calcul de la disparité

À partir d'images provenant de deux caméras synchronisées et observant la même scène sous deux angles différents, on construit une scène 3D par une méthode d'appariement des points des

images. En effet, un point de la scène n'aura pas la même position sur la paire d'images. On suppose que les caméras sont parfaitement alignées, ce qui implique qu'un point aura la même ordonnée au niveau des deux caméras. On appelle disparité le décalage au niveau des abscisses. Plus un point est éloigné des caméras, moins la disparité qui lui est associée sera importante. On va déterminer pour chaque primitive de l'image droite la primitive de l'image gauche qui lui serait le mieux appariée. Pour cela, on calcule un score de corrélation s à partir des moyennes gauche, m_g, et droite, m_d, au point considéré :

$$m_d(x_d, y_d) = \frac{1}{15} \sum_{j=-1}^{1} \sum_{i=-2}^{2} I_d(x_d + i, y_d + j), \quad m_g(x_g, y_g) = \frac{1}{15} \sum_{j=-1}^{1} \sum_{i=-2}^{2} I_g(x_g + i, y_g + j) \quad (1)$$

$$s = \sum_{j=-1}^{1} \sum_{i=-2}^{2} ||I_d(x_d + i, y + j) - m_d(x_d, y)| - |I_g(x_g + i, y + j) - m_g(x_g, y)|| \quad (2)$$

Plus le score obtenu est bas, plus la corrélation entre les deux points est forte. Pour chaque primitive, il s'agit donc de trouver le point donnant le score minimal.
En pratique, la disparité ne dépasse pas $\frac{N}{2}$ et elle est par ailleurs toujours positive. Une primitive d'abscisse i devra donc se corréler avec l'une des primitives d'abscisse $[i, i + \frac{N}{2}]$, et par ailleurs il n'est pas nécessaire de tenter une corrélation entre deux primitives dont le signe de gradient est différent. Le voisinage de corrélation est de 5 en abscisse et 3 en ordonnée, soit 15 pixels. On suppose, comme précédemment, que l'on a au maximum $N \times 5\%$ de primitives par image. Pour chaque primitive de l'image droite, il faut calculer S avec $5\% \times \frac{N}{2}$ primitives de l'image gauche. Le calcul de score avec une primitive de l'image gauche se fait en $O(30)$. Par conséquent, pour le calcul de tous les scores au niveau d'une primitive de l'image droite, on obtient une complexité en $O(15) + 5\% \times \frac{N}{2} \times O(30)$. Par ligne, le calcul de disparité pour un signe de gradient se fait en $O((5\% \times N) \times (O(15) + (5\% \times \frac{N}{2}) \times O(30))$. La complexité totale est donc de $L \times 2 \times O((5\% \times N) \times (O(15) + (5\% \times \frac{N}{2}) \times O(30))$. On récupère au maximum $10\%(2 \times 5\%)$ de points sur la carte de disparité noté D.

2.3. Filtrage de la carte de disparité

Le filtrage de la carte de disparité est similaire à celui du filtrage des primitives. Le voisinage de comptage est un rectangle 5×3 autour du point étudié. On note n le nombre de points à disparité non nulle de la zone, t une constante de tolérance et on calcule la somme s en ce point $s = \sum_{j=-1}^{1} \sum_{i=-2}^{2} D(x_d + i, y_d + j)$. Si $|D[x_d + i, y_d + j] - s/n| < (s/n) \times t$, le point est conservé. La complexité totale est de $O(L \times 15 \times N \times 10\%)$.

2.4. Profil de la route

Cette application visant à détecter les obstacles se trouvant sur la route, il faut pouvoir déterminer la route proprement dite, ou plus précisément la disparité de chacun des points de la route, ce qui peut être fait avec la carte de disparité D. Dans le cas d'une route plane, on sait que, sur une même ligne de D, tous les points de la route ont la même disparité. On peut donc considérer que pour chaque ligne la route possède une certaine disparité. De cette manière, on considère l'espace ayant les lignes pour ordonnée et les disparités pour abscisse et dans lequel plusieurs points d'une ligne y ayant la même disparité δ s'accumulent au point (y, δ). C'est l'espace de v-disparité présenté dans la figure 2. Dans le cas d'une route plane, la route correspond dans cet espace à une droite. On peut obtenir l'équation de cette droite en utilisant la transformée de Hough [3], ce qui permet de calculer la disparité de la route en chaque ligne. En ce qui concerne le roulis, le phénomène se ressent au niveau de la carte de la v-disparité par une droite

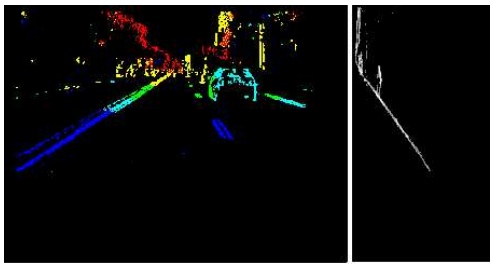


FIG. 2 – Carte de la v-disparité.

représentative de la route plus épaisse. La recherche de l'équation en devient plus ardue. L'idée est de faire subir le roulis inverse à la route pour obtenir. sur la carte une droite bien définie. Le roulis étant lié à l'environnement, on utilise une méthode exhaustive pour en rechercher la valeur. Sachant que le roulis se situe entre -3° et 3°, on fait subir à la carte de v-disparité 30 roulis (dans cet intervalle), ainsi 30 espaces de Hough sont obtenus. Celui possédant le maximum des maxima de tous les espaces de Hough possédera le roulis exact[8]. L'application doit aussi être robuste aux problèmes de vrillage. Celui-ci se ressent par une route correspondant plus à une jonction de droites dans la carte de v-disparité. L'idée apportée par le LIVIC est de découper cette carte en 5 tranches verticales de même taille et d'y effectuer indépendamment les calculs, d'où 5 espaces de Hough. En tenant compte du roulis et du vrillage, 150 espaces de Hough doivent être calculés. Lors du calcul de la disparité de la route, on récupère les 5 maxima des tranches des différents roulis. On a au maximum 10% de points sur la carte de disparité, ce qui nous donne une complexité en $O(L \times (10\% \times N) \times 30)$ pour chaque roulis, donc $O(30 \times L \times (10\% \times N) \times 30)$ pour la détermination du profil de la route.

2.5. Amélioration de la carte de disparité

Maintenant que l'on connaît le profil de la route, on peut améliorer la carte de disparité en corrigeant les erreurs d'appariement commises précédemment[7]. On va ré-effectuer un calcul de corrélation, à la différence qu'à présent on suppose qu'on connaît la disparité de chacune des primitives. On prend une primitive i de l'image droite et on lui associe une disparité δ égale à la disparité de la route sur sa ligne. Si ce point arrive à se corréler dans un voisinage 5×5 autour du point $i + \delta$ dans l'image gauche, alors ce point appartient à la route. Le cas échéant, on le considère comme un obstacle. Le calcul de corrélation se fait toujours sur un voisinage 5×3, soit 15 pixels. En pratique on constate que 10 points sur 25 sont testés par primitive. La complexité est donc en $O(L \times (10\% \times N) \times 10 \times 15)$.

2.6. Extraction de l'espace roulable

On a à présent une carte de disparité nous indiquant la nature des primitives représentées (route ou obstacle). L'idée est de parcourir cette carte de bas en haut pour déterminer l'espace roulable. Le traitement se fait par le bas pour éviter de considérer que ce qui se trouve devant un véhicule-obstacle comme de l'espace libre. La complexité de cette étape est en $O(L \times N)$. Le résultat du traitement est représenté sur la figure 3.

3. Contraintes

Les contraintes initiales de l'application définissent la latence du traitement, le débit des images et leur la taille. La latence du traitement d'une paire d'images doit être inférieure à 100 ms. En

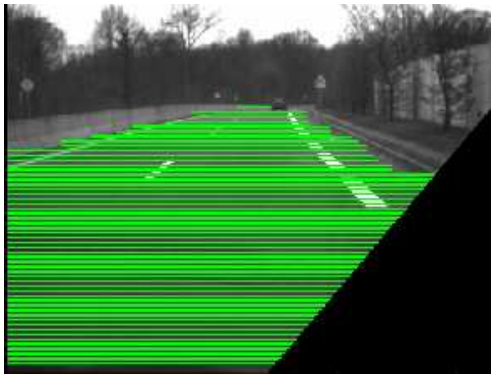


FIG. 3 – Espace roulable.

revanche, le débit de sortie des images au niveau des deux caméras est de 10 ms. Enfin, la taille des images a été initialement fixée à une résolution 1024×1024 pour 16 bits par pixel, soit 65536 niveaux de gris possibles. Leur taille (individuelle) est donc de 2 Mo.
Par ailleurs, pour que le coût du circuit soit acceptable par les équipementiers, on considère un nombre de broches de 512 maximum, une fréquence de fonctionnement de 200 MHz et une mémoire interne ne dépassant pas 1 Mo. Plusieurs conséquences peuvent se déduire de ces quelques contraintes :

– La latence permise est bien supérieure au débit. On voit rapidement qu'il n'est pas possible d'effectuer la totalité du traitement en 2 millions de cycles. Cela impose donc que la puce est constituée de plusieurs *clusters* (au maximum 10) qui fonctionnent en parallèle et décalé d'une image dans le temps, comme illustré figure 4.

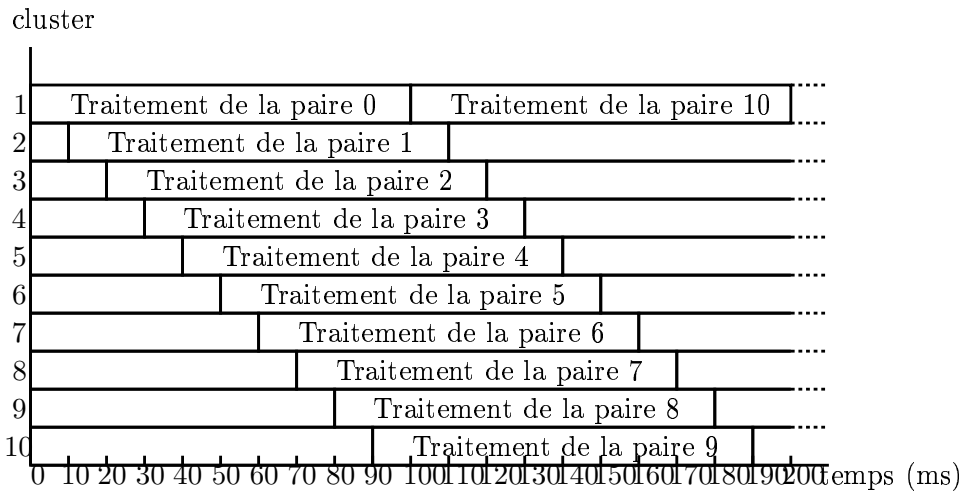


FIG. 4 – Exécution sur 10 *clusters*.

– Les images occupent chacune 2 Mo, soit 4 Mo pour une paire stéréoscopique soit 40 Mo si l'on a 10 *clusters*. Leur stockage ne peut pas se faire dans la mémoire interne de la puce, étant données les contraintes d'intégration. Les images sont donc à stocker en mémoire externe. Le nombre de broches étant limité à 512, on considère qu'une seule RAM externe va être

utilisée. Le schéma de la figure 5 présente l'architecture d'une puce dont les sous-systèmes (indépendants) gèrent les différentes paires stéréoscopiques provenant de la RAM externe.

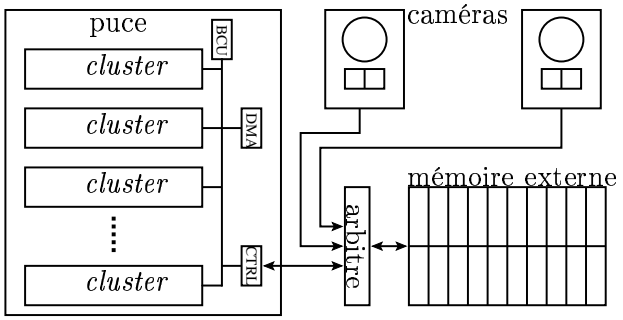


FIG. 5 – Architecture globale du système.

– pour chaque pixel de chaque image, la RAM externe subit une écriture effectuée par une des caméras et deux lecture par la puce. Elle contiendra 20 images de 2 Mo (10 paires stéréoscopiques). Elle possède une latence de 12 cycles et peut transférer jusqu'à 32 mots en une rafale.

On peut à présent estimer la charge au niveau de la RAM externe. En 100 ms, chaque caméra va copier dix images et la puce va lire 10 paires stéréoscopiques, soit 20 images, et ce deux fois. On a donc 60 transferts d'images. Chaque image contient 1024 lignes et chaque ligne se découpe en 16 paquets de 64 pixels. Un transfert d'image se fait en 16384 paquets et les 60 en 983040 paquets. La puce tourne à 200 MHz, ce qui nous laisse 20 millions de cycles (100 ms) pour transférer les 983040 paquets, soit 21 cycles par paquet, ce qui est insuffisant (un paquet requière 44 cycles minimum pour être transféré). La solution à ce problème est de réduire les spécifications, et plus précisément la résolution des images, passant de $1024 \times 1024 \times 16$bpp à $1024 \times 256 \times 16$bpp. Le nombre de paquets passe ainsi de 983040 à 245760 et on a à présent 82 cycles par paquet. Ce choix est celui qui permet de conserver les meilleurs propriétés à l'application pour cette quantité de mémoire.

4. Méthode d'intégration

La puissance de calcul nécessaire impose l'utilisation d'une architecture parallèle pour chaque *cluster*. Les fonctions à effectuer sont par contre très différentes les unes des autres, donc c'est vers une implantation MIMD que nous nous orientons. Afin de permettre une modification tardive de l'application, voire de permettre des mises à jour, nous optons pour une architecture multiprocesseur à mémoire partagée homogène. Chaque *cluster* possède p processeurs MIPS R3000 reliés par un bus et tout le traitement est effectué par logiciel.

Nous utilisons l'approche préconisée par l'environnement de conception Disydent [1].

– l'application est réécrite sous forme de tâches parallèles échangeant des données par l'intermédiaire de files de taille finie et bloquantes (graphe de processus de Kahn [5] restreint [9]). Cette phase permet de mettre en évidence le parallélisme ou l'aspect *pipeline* de l'application. De plus, les propriétés des graphes de Kahn sont telles que le comportement de l'application ne dépend ni de l'assignation des tâches aux processeurs, ni de leur ordonnancement (si les files sont de taille suffisante) ;

– l'application parallèle est alors implantée sur une architecture multiprocesseur sur laquelle s'exécute un micro-noyau symétrique [10] ;
– l'architecture est ensuite simulée avec un simulateur dont les modèles sont précis au bit et au cycle près, et les performances sont mesurées ;
– si les performances ne sont pas satisfaisantes, alors les tâches qui posent problèmes sont réalisées en matériel avec un l'outil de synthèse d'architecture à partir du code source de la tâche (qu'il faut restructurer assez profondément pour que le résultat de la synthèse soit efficace).

La spécification parallèle est structurée de manière identique à la version séquentielle (c.f. figure 6). On peut noter que l'application se découpe en deux pipelines. La raison est que la détermination de l'équation de la disparité de la route ne peut se faire que si on possède les informations de chaque ligne. Il nous faut donc terminer le pipeline à cette étape là, calculer la disparité de la route et commencer le second pipeline. Vu que celui-ci nécessite les pixels des images, il nous faut à nouveau rapatrier les deux images, d'où un total par image de deux lectures provenant de la puce.

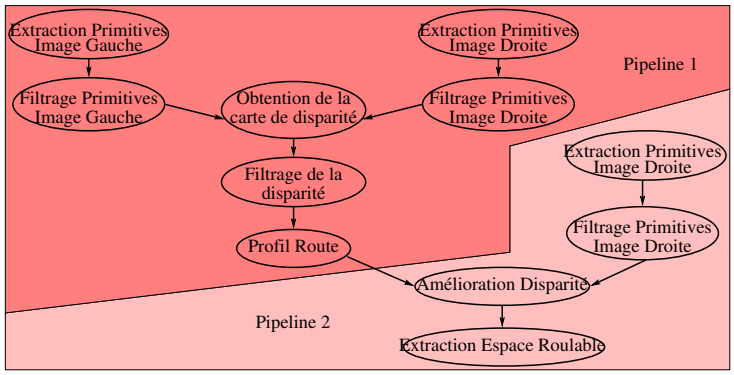


FIG. 6 – Application pipelinée

Notre application étant un pipeline, chaque étage a droit à 100 ms (latence pour une paire stéréoscopique), soit 20 millions de cycles. On a 256 lignes, donc 256 entrées. On peut donc considérer qu'en moyenne, chaque étage possède un maximum de $20000000/256 = 78125$ cycles par ligne. Les temps de calcul de chaque processeur et l'occupation du bus doivent être inférieures à cette valeur pour que l'implantation soit viable. Cette moyenne va surtout nous permettre de savoir si une tâche doit être modifiée, logiciellement ou matériellement.

L'étude du coût de chaque tâche sur cette architecture est détaillée dans [12]. Nous nous contentons d'en donner ici les résultats quantitatifs, pour des lignes de cache de taille identiques à la largeur de l'image (32 mots) et une latence de 40 cycles pour la récupération d'une ligne de la mémoire embarquée dans la puce. La récupération d'un paquet de 32 mots de la mémoire externe prend environ 120 cycles.

– le rapatriement des lignes d'images est effectué grâce à un DMA, et coûte 4% de la bande passante par paire de lignes. On fait l'hypothèse que ce transfert à lieu pendant que les processeurs exécutent par ailleurs des calculs utiles ;
– l'extraction des primitives prend 17000 cycles et fournit un tableau de bit dont au plus 5% sont à 1. La bande passante utile est de 1,5% ;
– le filtrage des primitives est optimisé en détectant au plus tôt qu'un mot ne contient aucun bit à 1. Il faudra 33300 cycles pour filtrer les gradients positifs et les gradients négatifs, et cela

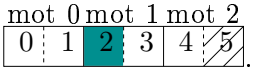
occupera moins de 1% de la bande passante ;
- le calcul de la disparité s'effectue en ayant une ligne de l'image gauche, une ligne de l'image droite ainsi que les gradients de même signe des deux lignes. Le nombre de cycles nécessaire à ce calcul est de 283200 cycles, et il faut 95200 cycles de bus pour échanger les données. Ceci ramené aux 78125 cycles dont on dispose fait que le processeur exécutant cette tâche devrait tourner à 725 MHz, et que le bus est occupé à plus de 120% ! On choisit d'utiliser un coprocesseur se chargeant d'effectuer tout le calcul. Le coprocesseur récupère les lignes de pixels et de gradients. En sortie, on obtient $5\% \times 1024$ mots (mot = abscisse + disparité). Ce composant offre plusieurs avantages. Les lignes étant rapatriées une unique fois, la charge du bus s'en trouve réduite. On évite ainsi la plupart des *miss* engendrés lors du calcul. De plus, la parallélisation des calculs au niveau matériel permet une réduction du nombre de cycles pour les calculs de moyenne et de score ;
- le filtrage de la carte de disparité utilise 17200 cycles CPU et 5% de la bande passante ;
- le profil de la route est déterminé par des transformées de Hough. Le problème est que la gestion du roulis implique le calcul de 150 transformées en parallèle, ce qui n'est possible ni du point de vue des échanges avec la mémoire, ni du point de vue des unités de calcul. Après discussion, il apparaît que la précision apportée par ce calcul gigantesque n'est pas absolument nécessaire pour la détection d'obstacles. On décide donc de se ramener à un unique calcul en faisant l'hypothèse que la voiture est stable. Le nombre de cycle CPU est alors de 26600 cycles et le taux d'occupation du bus de 17%.
- l'amélioration de la carte de disparité nécessite un profil de la route complet, donc il faut avoir traité l'ensemble des lignes avant de pouvoir exécuter cette étape. De plus, il faut parcourir de nouveau les images stockées en mémoire externe. Le nombre de cycles CPU est 161800 et le taux d'utilisation du bus est de 48% ;
- l'extraction de l'espace roulable travaille sur une liste de couples (abscisse, disparité) contenant en moyenne 10% du nombre total de points par ligne, soit un stockage en mémoire sur 100 mots. L'occupation du processeur est de 8400 cycles et celle du bus négligeable (0,2%).

5. Description du coprocesseur

Les calculs et les échanges induits par le calcul de la disparité font que l'usage de matériel *ad-hoc* est inévitable. Nous décrivons à présent l'architecture retenue pour implanter ce calcul.
Les données en entrée du coprocesseur étant des mots et les pixels étant des demi-mots, des optimisations ont été effectuées pour minimiser le trafic avec la mémoire externe. Les moyennes et le score se calculent de la même manière pour les deux pipelines, excepté sur le fait que le pipeline 1 travaille avec 3 lignes (15 pixels) et le pipeline 2 avec une ligne (5 pixels). Pour la suite, les explications se feront avec ce dernier car cela est plus simple à expliquer.

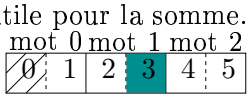
5.1. Calcul de moyenne

Cette étape consiste à calculer la moyenne sur un voisinage de 5 pixels autour du point étudié. La gestion des données (deux pixels par mot) permet d'effectuer la moyenne en 4 cycles (3 cycles de somme + 1 cycle de division). Dans le pipeline 1, l'indépendance des lignes permet le respect de ces temps (parallélisation des opérations). La parité du pixel dans le mot oblige à considérer deux cas pour les trois premiers cycles du calcul (la case grise correspond au pixel sur lequel on effectue le calcul de la moyenne) :

- si le pixel étudié est d'indice pair, le voisinage est le suivant : 

. Au premier cycle, on peut sommer deux pixels. En effet, on récupère le mot 0 contenant deux pixels

nécessaires à ce calcul. Au deuxième cycle, il se passe la même chose avec le mot 1. Au troisième cycle, seul le demi-mot de poids fort du mot 2 est utile pour la somme.

- si le pixel étudié est d'indice impair, le voisinage est le suivant : 

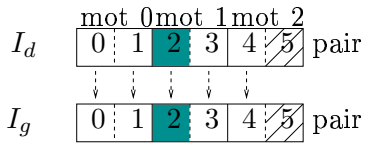
. Au premier cycle, on ne somme que le demi-mot de poids faible du mot 0. Au deuxième cycle, les deux pixels du mot 1 sont à récupérer. Au troisième cycle, on fait la même chose avec le mot 2.

La somme étant calculée, il faut diviser par 15 pour le pipeline 1. Cette division est effectuée à l'aide du développement limité d'ordre 3 : $\frac{p}{15} \approx p >> 4 + p >> 8 + p >> 12$. Pour le pipeline 2, la division par 5 est effectuée ainsi : $\frac{p}{5} \approx \frac{51 \times p}{256} = ((p + p + p) << 4) + (p + p + p)) >> 8$.

5.2. Calcul de score

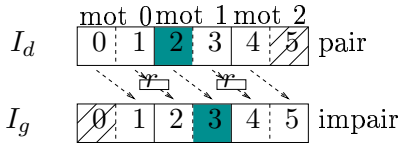
Cette étape consiste à calculer le score de corrélation entre deux pixels (c.f les équations 1 et 2). Les problèmes dus à la différence de typage se ressentent aussi lors de cette étape. Il faut à présent prendre en compte la "parité" des deux pixels simultanément et donc étudier les 4 cas pair/pair, pair/impair, impair/pair, impair/impair. Les figures suivantes montrent les appariements à effectuer entre pixels. On note $x \perp y$ le fait que l'on soustrait le pixel x de I_d du pixel y de I_g et que l'on en prend la valeur absolue.

pair/pair :



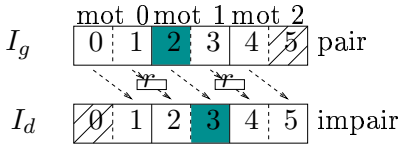
Ce cas ne pose pas de problème réel, vu que les soustractions se font deux à deux à chaque cycle : premier cycle $0 \perp 0$ et $1 \perp 1$, deuxième cycle $2 \perp 2$ et $3 \perp 3$, troisième cycle : $4 \perp 4$.

pair/impair :



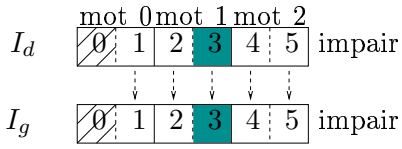
Dans ce cas, les opérandes de chaque soustraction ne possèdent pas des indices de même parité. On remarque que le bit de poids faible du mot 0 de droite (pixel 1) doit être soustrait au bit de poids fort du mot 1 de gauche (pixel 2), ce qui pose problème puisque deux mots d'une même ligne ne peuvent pas être accédés au même cycle. Il faut donc sauvegarder le pixel 1 de droite dans un registre (r) et l'utiliser au cycle suivant. Le procédé est le suivant : premier cycle $0 \perp 1$ et sauvegarde 1 dans r, deuxième cycle $r \perp 2$ et calcul $2 \perp 3$ et sauvegarde 3 dans r, troisième cycle $r \perp 4$ et calcul $4 \perp 5$.

impair/pair :



Ce cas là se présente de la même manière que précédemment. Comme la soustraction se fait avec une valeur absolue, l'ordre des opérandes n'est pas fixé et l'on peut échanger x et y. Le procédé est le suivant : premier cycle $0 \perp 1$ et sauvegarde 1 dans r, deuxième cycle $r \perp 2$ et calcul $2 \perp 3$ et sauvegarde 3 dans r, troisième cycle $r \perp 4$ et calcul $4 \perp 5$.

impair/impair :



Ce cas ne pose pas de problème, la parité des opérandes de chaque soustraction étant identique : premier cycle $1 \perp 1$, deuxième cycle $2 \perp 2$ et $3 \perp 3$, troisième cycle $4 \perp 4$ et $5 \perp 5$.

Avec une telle architecture, un score de disparité est calculé en approximativement 10 cycles pour des images très fournies. Lorsqu'un pixel ne contient pas de primitive (grâce à la phase de filtrage préalable), la décision est prise au plus tôt et améliore les performances. Le pipeline 1 et le fort degré de parallélisation des calculs impose l'utilisation d'une vingtaine d'opérateurs effectuant la valeur absolue d'une soustraction de deux opérandes. Quant à la mémoire interne

de ce coprocesseur, chaque ligne de pixels occupe 512 mots et chaque ligne de gradients 32 mots, la ligne de disparité en sortie étant de 128 mots. On a 6 lignes de pixels, 2 lignes de gradients, ce qui nous donne $6 \times 2048 + 2 \times 128 + 512 \approx 13$Ko de mémoire interne. Le coprocesseur a été synthétisé avec l'outil de synthèse d'architecture UGH [2]. Il contient 300,000 portes.

6. Architecture de test

Les simulations sont effectuées sur un *cluster* et non pas sur la puce entière pour des raisons de vitesse de simulation. L'architecture du *cluster* est composée de 7 processeurs avec un cache instruction de 128 lignes de 16 mots et un cache de données de 32 lignes de 8 mots. Le bus est un PI-Bus. Les composants possèdent une interface VCI et son connectés au bus par l'intermédiaire de *wrappers* (non représenté sur la figure 7). Le coprocesseur est connecté au bus par l'intermédiaire de files matérielles VCI, dont les pilotes logicielles supportent la sémantique de communication définies par Kahn [4]. L'architecture du *cluster* est basée sur un bus, mais il n'est pas possible de bénéficier de la localité des échange pour espionner le bus pour assurer la cohérence mémoire car nous ne l'accédons qu'à travers l'interface VCI. Les deux composants

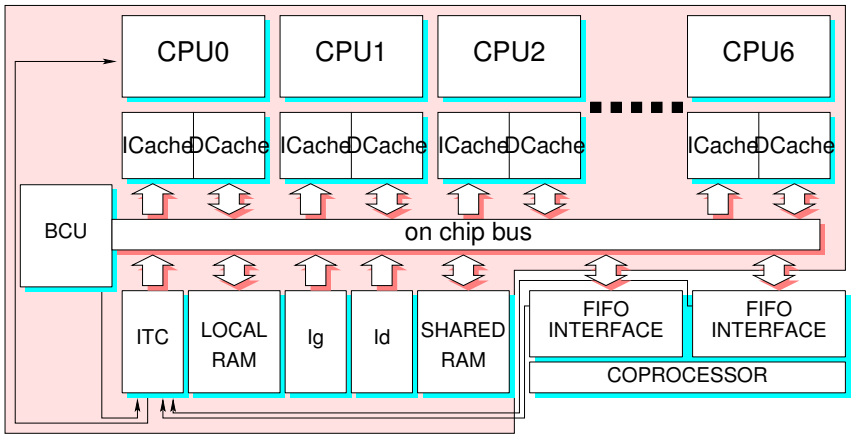


FIG. 7 – Architecture d'un *cluster* pour la validation.

FIFO sont reliés à l'ITC (contrôleur d'interruptions). Dès que le coprocesseur veut lire ou écrire une donnée dans une FIFO mais que cela lui est impossible, il génère (*via* l'une des files) une interruption vers l'ITC. Celui-ci génère à son tour une interruption vers le processeur 0 qui se charge de la gestion de l'interruption (rôle du noyau) afin de débloquer le demandeur (dans notre cas un coprocesseur). La RAM locale contient les zones locales à chaque processeur. La RAM partagée contient les structures de communication inter-threads, et est accédée sans passer par le cache. Les RAMs I_g et I_d stockent les images de la paire et sont dans l'espace caché afin de bénéficier des rafales.

La simulation de l'application sur le *cluster* multiprocesseur à lieu sous le simulateur CASS [11]. Il exécute des modèles décrits en C, de niveau RTL sous forme d'automates. Les déclarations et instanciations des différents composants utilisés dans l'architecture et leurs connexions sont faites en VHDL.

La vérification des résultats se fait par comparaison avec les résultats obtenus lors de la validation de cette même application sous Linux. En ce qui concerne le premier pipeline, on récupère

les coordonnées du maximum dans l'espace de Hough (le couple de coordonnées polaires correspondant à la droite de disparité de la route) ainsi que le nombre de votants pour ce point (la valeur accumulée en ce maximum). Au niveau du second pipeline, la technique utilisée a été d'additionner les abscisses de chacune des lignes correspondant à l'espace roulable dans une unique variable. Ces informations constituent en quelque sorte la « signature » des différentes étapes de l'application. La comparaison des informations avec celles issues de l'exécution native déterminera la validité de l'application sur la puce. Cette méthode peut paraître archaïque mais si les résultats rendus sont corrects pour les 164 simulations, on peut considérer à juste titre que l'application est validée.

Image gauche. Image droite Espace roulable

FIG. 8 – Exemple de détermination de l'espace roulable.

Le traitement de la paire d'images pire cas de notre échantillon prend 30325607 cycles avec une charge du bus de 81.8% sur cette architecture[2]. Le résultat d'une simulation est présentée dans la figure 8.
Les contraintes posées au départ font que la simulation doit s'effectuer en au maximum 20 millions de cycles. Les résultats obtenus sur 164 tests montrent que le traitement se fait entre 30 et 32 millions de cycles (160% de ce qui était escompté). L'analyse que nous en faisons est la suivante :
- Lors de l'étude, la synchronisation entre les tâches n'a pas été prise en compte ;
- Il est possible que le nombre d'instructions estimé lors de l'étude ait été optimiste par rapport au nombre de cycles des instructions dépendant de la compilation ;
- Lorsqu'une tâche est bloquée sur une file, le noyau prend la main et fait de l'attente active sur une variable non cachée. Les tâches dont le traitement est rapide et qui ne font qu'attendre des données font que cette attente active devient significative au niveau du bus et le surcharge ;
- Les communications avec la mémoire externe ont été considérées durant l'étude comme effectuées *via* un DMA, hors la mise en œuvre de son utilisation reste à faire.

Toutes ces raisons ont pu amener à cette augmentation de 60% du temps de simulation. Une analyse plus détaillée reste à faire pour identifier le facteur le plus significatif.

7. Conclusion

Cette étude est un premier pas vers l'intégration d'une application de vision stéréo pour la détection d'obstacles.

[2] Le temps de simulation est de 370 s, soit 81961 cycles/s pour 23 composants instanciés

L'étude préalable à mis en évidence un problème grave : la limitation du nombre de broches ne permet pas d'avoir une bande passante assez importante vers la mémoire, ce qui nous oblige à mémoriser beaucoup d'information dans la puce, et impose un parallélisme de traitement important. Une solution à ce problème pourrait être de compresser les données, mais cela doit se faire sans pertes (l'effet de *blocking* est par exemple très néfaste), et cela coûte encore plus en terme de puissance de calcul. Un autre alternative serait d'utiliser des liens série à très haute fréquence, avec potentiellement les problèmes d'intégration que cela induit.
La méthode d'intégration a permis d'obtenir assez rapidement des résultats de simulation, dont on a vu qu'ils ne sont pas à la hauteur de ce que l'on espérait. Un raffinement de l'architecture est donc nécessaire, notamment vis-à-vis de certains transferts de données qui pourraient être effectués beaucoup plus efficacement avec des DMA. Pour finir, l'utilisation de coprocesseurs spécialisés est une nécessité pour obtenir de la performance à faible coût, une fois la conception initiale effectuée.

Bibliographie

1. Augé (I.), Pétrot (F.), Donnet (F.) et Gomez (P.). – Intégration sur plate-forme matérielle/logicielle de spécifications « C » parallèles. *Annales des télécommunications*, vol. 59, n 7-8, juillet-août 2004.
2. Donnet (F.). – *Synthèse de Haut Niveau Contrôlée par l'Utilisateur.* – LIP6, Thèse de PhD, Univ. P. et M. Curie, janvier 2004.
3. Duda (R. O.) et Hart (P. E.). – Use of the hough transform to detect curves and lines in pictures. *Communications of the ACM*, vol. 15, n1, 1972, pp. 11–15.
4. Hommais (D.), Pétrot (F.) et Augé (I.). – A practical toolbox for system level communication synthesis. *In : 9th Int. Symp. on Hw/Sw Co-design*, pp. 48–53. – avril 2001.
5. Kahn (Gilles). – The semantics of a simple language for parallel programming. *In : Information Processing 74*, pp. 471–475. – Stockolm, Suède, août 1974.
6. Labayrade (R.). – *Détéction Générique, Robuste et Rapide d'Obstacles Routiers par Stéréovision Embarquée.* – Paris, France, Thèse de PhD, Univ. P. et M. Curie, janvier 2004.
7. Labayrade (R.) et Aubert (D.). – In-vehicle obstacles detection and characterization by stereovision. *In : IEEE In-Vehicle Cognitive Computer Vision Systems.* – Graz, Austria, April 2003.
8. Labayrade (R.) et Aubert (D.). – A single framework for vehicle roll, pitch, yaw estimation and obstacles detection by stereovision. *In : IEEE Intelligent Vehicles Symposium Proceedings.* – Columbus, OH, juin 2003.
9. Parks (T. M.). – *Bounded Scheduling of Process Networks.* – Thèse de PhD, Electronics Research Laboratory, Berkeley, 1995.
10. Pétrot (F.), Gomez (P.) et Hommais (D.). – Lightweight implementation of the posix threads api for an on-chip mips multiprocessor with vci interconnect. *In : Embedded Software for SoC*, éd. par Jerraya (A. A.), Yoo (S.), Verkest (D.) et Wehn (N.), 1 3, pp. 25–38. – Kluwer Academic Publisher, novembre 2003.
11. Pétrot (F.), Hommais (D.) et Greiner (A.). – A simulation environment for core based embedded systems. *In : 30th Int. Simulation Symp.*, pp. 86–91. – Atlanta, Georgia, avril 1997.
12. Zaoui (D.). – *Exploration architecturale pour l'intégration d'une application de stéréovision dans un système sur puce multiprocesseur.* – Rapport de dea, Université Pierre et Marie Curie, septembre 2004.