Université de Montréal

Réseaux de neurones à faible précision et réalisation sur ordinateur à logique programmable

par

Steven Pigeon

Département d'informatique et de recherche opérationelle
faculté des Arts et Sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès science (M.Sc)
en informatique

mars 1996

Université de Montréal
Falculté des études supérieures

ce mémoire intitulé

Réseaux de neurones à faible précision et
réalisation sur ordinateur à logique programmable

présenté par

Steven Pigeon

a été évalué par un jury composé des personnes suivantes:

El Mostapha Aboulhamid
Yoshua Bengio
Jocelyn Cloutier

Mémoire accepté le:.....

Résumé

Dans ce mémoire, nous explorons les aspects logiciels et matériels des réseaux de neurones artificiels. Nous examinons les propriétés de stabilité numérique, tels que le nombre de bits nécessaires pour représenter un poids, une activation et le gradient dans un réseaux de neurones artificiels, et comment on peut remplacer les opérations arithmétiques traditionnellement associées aux réseaux de neurones artificiels par des opérations matériellement moins coûteuses. Nous considérerons dans un second temps les aspect matériels des réseaux de neurones artificiels (l'ordinateur à logique programmable V.I.P. et la possibilité de l'apprentissage en matériel). Nous montrons qu'avec l'avènement des circuits à logique programmable (les *field programmable gate arrays*, ou FPGA) la distinction entre *logiciel* et *matériel* devient avantageusement floue rendant ainsi possible la réalisation de systèmes dédiés efficaces à l'aide d'*ordinateurs à logique programmable* à une fraction des coûts habituellement rencontrés.

Sommaire

Ces dernières années, en parallèle avec l'accroissement rapide de la puissance des ordinateurs, de nombreuses techniques d'intelligence artificielle ont bourgeonné. Les méthodes statistiques et les réseaux de neurones artificiels (et très souvent un mélange des deux) ont gagné l'intérêt des chercheurs pour une seule et bonne raison: les réseaux de neurones, loin d'être une solution illusoire, ont démontré des capacités inégalées dans certains domaines, comme en particulier la reconnaissance de caractères manuscrits, de la voix, et des séries temporelles en général. Le seul problème des réseaux de neurones, c'est qu'ils sont très voraces! Ils demandent beaucoup de puissance de calcul et les stations de travail actuellement disponibles sont loin d'offrir des performances intéressantes pour des applications concrètes. La nature essentiellement parallèle des réseaux de neurones peut faire paraître les ordinateurs massivement parallèles comme étant une solution miracle.

Cependant, la plupart des gens ne disposent toujours que de machines séquentielles ou faiblement parallèles comme outil de travail principal et les ordinateurs massivement parallèles demeurent encore hors de portée. Que ce soit par leur prix, la difficulté de la programmation ou par leurs réseaux de communication encore trop peu efficace par rapport à la puissance des processeurs, les ordinateurs massivement parallèles ne sont pas aussi accessibles que les stations de travail (qu'il s'agisse de Sun, SGI ou même de PC) qui forment l'habituel outil du chercheur en informatique. Il s'impose donc d'explorer deux avenues qui ne sont pas obligatoirement disjointes. Premièrement, l'aspect logiciel des réseaux de neurones. Il convient de se poser des questions fondamentales sur les propriétés de stabilité numérique, tels que le nombre de bits nécessaire pour représenter un « poids » et une « activation », et comment on peut

remplacer les opérations arithmétiques traditionnellement associées aux réseaux de neurones par des opérations matériellement moins coûteuses. Deuxièmement, à la lumière de la première voie, il faut se demander quelles seraient les propriétés idéales d'un matériel supportant les calculs d'un réseau de neurones. Quelle est la granularité des calculs que l'on réalise dans les réseaux de neurones? Est-ce que les algorithmes classiques peuvent nous venir en aide? Quelles sont les erreurs que l'on peut introduire dans le calcul sans que la performance des réseaux de neurones artificiels ne s'en ressente? Quels seront les modèles de parallélisme utilisables et sous quelles hypothèses? Quelles sont les performances que l'on doit attendre du parallélisme? Quels sont les coûts que l'on doit se préparer à rencontrer? Quel est le seuil de rentabilité des technologies alternatives?

C'est essentiellement ces questions qui motivent le présent mémoire. Dans un premier temps, nous allons explorer les aspects algorithmiques et logiciels (l'apprentissage avec des nombres à point fixe de faible précision, convolutions rapides, etc.) et dans un second les aspect matériels (l'ordinateur à logique programmable V.I.P. et l'apprentissage en matériel). Nous allons démontrer qu'avec l'avènement des circuits à logique programmable (les *field programmable gate arrays*, ou FPGA) la distinction entre *logiciel* et *matériel* devient avantageusement floue rendant ainsi possible la réalisation de systèmes dédiés efficaces à l'aide d'*ordinateurs à logique programmable* à des coûts raisonnables.

Table des matières

Table des illustrations

À mes parents, Frances et Jean-Noël,
pour leur soutien indéfectible

Remerciements

J'aimerais remercier Éric Cosatto, de Bell Labs, Holmdel, pour son support technique lors de l'été 1995 où nous avons conçu et réalisé la carte V.I.P., et bien entendu, pour toute l'aide qu'il a continué de nous donner par la suite. Dire que son aide a été précieuse relève de l'euphémisme. Eduard Säckinger, aussi de Bell Labs, m'a fait parvenir la description complète du LeNet 1a et a répondu patiemment à toutes mes questions, mêmes celles dont j'aurais dû connaître la réponse, ce pour quoi je ne puis qu'être reconnaissant. Je ne peux évidemment pas oublier Jocelyn Cloutier pour l'aide précieuse qu'il m'a apportée tout au long de ce projet tout en me laissant libre de mes choix.

Chapitre 1: Introduction.

Depuis vingt ans il a été observé qu'à tous les dix-huit mois, environ, le matériel disponible double en puissance de calcul. Qu'ils utilisent des méthodes classiques d'intelligence artificielle ou de nouvelles méthodes, comme les méthodes statistiques ou encore les réseaux de neurones, les utilisateurs poussent toujours le matériel à sa limite.

Et les réseaux de neurones illustrent bien cette situation; ils occupent une place de plus en plus importante dans le domaine de l'intelligence artificielle et ils sont appliqués à toutes les sauces. Mais ils sont très voraces en temps de calcul. Même l'évaluation (ce qui est habituellement nommé « passe avant ») reste encore dans certains cas assez complexe pour qu'il soit impossible d'obtenir des résultats en temps satisfaisant sur des ordinateurs personnels relativement puissants. Nous verrons aussi que les ordinateurs actuels ne sont pas très bien adaptés à ce genre de tâche, et nous verrons comment nous pouvons améliorer le logiciel et le matériel pour résoudre ces problèmes très efficacement.

Pour la rédaction du présent mémoire, nous nous sommes d'abord penchés sur l'aspect logiciel des réseaux de neurones, en particulier ceux du type LeNet à base de convolution, et nous présenterons les algorithmes que l'auteur a obtenus pour les convolutions rapides. Nous avons ensuite considéré un modèle numérique matériellement réalisable à peu de frais pour supporter les réseaux de neurones. Nous montrerons dans ce mémoire que pour les réseaux de neurones à faible précision, l'apprentissage peut être qualifié de plus lent mais possible. Nous décrirons ensuite le matériel nécessaire pour supporter l'évaluation en exploitant un parallélisme élevé pour la réalisation rapide des convolutions. Pour le présent mémoire, l'auteur a développé le contrôleur systolique, pour lequel il présente un assembleur, et un simulateur fidèle à la carte V.I.P. — le *Virtual Image Processor* — un *debugger* et quelques autres

outils qui complètent un noyau d'environnement de développement. Nous avons aussi mené, pour ce mémoire, des expériences visant à comparer les différents modèles d'apprentissage (précision arbitraire, faible précision et mixte).

Les premiers chapitres situent et introduisent les notions pertinentes à nos travaux alors que les derniers chapitres présentent ceux-ci ainsi que les résultats obtenus. Nous discuterons des notions de base pour les réseaux de neurones, l'évaluation et l'apprentissage (par l'algorithme de la rétropropagation de l'erreur) et quelques autres notions sur les réseaux, telles que les *cartes*. Nous présenterons notre choix de matériel, les circuits à logique programmable, et nous verrons comment ils présentent une solution assez performante à un prix raisonnable. Nous considérerons ensuite une solution numérique réalisable à peu de frais en matériel comme en logiciel, et nous présenterons les modifications à l'algorithme de rétropropagation pour accommoder le système arithmétique à faible précision. Nous étudierons une application typique en reconnaissance de caractères manuscrits, les réseaux LeNet, que nous présenterons dans le détail dans l'annexe 1. Nous explorerons et comparerons les solutions logicielles et matérielles pour accélérer le calcul des convolutions, qui forme la plus grande partie du calcul dans un réseau LeNet et finalement nous décrirons l'implantation d'une machine à convolution rapide sur un ordinateur à logique programmable. Au chapitre 5 nous présenterons les résultats obtenus. Pour conclure nous explorerons d'autres solutions, y compris l'implantation sur silicium des algorithmes étudiés, et nous discuterons des points importants que nous avons dégagés dans ce mémoire.

1.1 Problèmes en intelligence artificielle.

D'un point de vue pragmatique, l'intelligence artificielle se divise en deux catégories principales. Il y a l'intelligence artificielle où l'on sait comment résoudre les problèmes et les solutions sont de nature algorithmique et systématique (fouille de l'espace d'état, systèmes génératifs, programmation logique, systèmes experts, optimisation, etc.), et il y a celle où la connaissance sur les problèmes est insuffisante ou la complexité trop grande pour mener directement à une solution; il faut *apprendre* — et bien entendu il y a cette zone grise où les méthodes classiques et les méthodes nouvelles se confondent.

Dans le cas où le problème est relativement bien compris et où l'expertise humaine est disponible, le chercheur considérera les différents paradigmes classiques pour choisir celui qui lui convient le mieux. Par exemple, s'il s'agit d'un problème de taxonomie, il y a de fortes chances que ce soit le modèle du système expert qui soit choisi. Dès lors, on extrait la connaissance de l'expert humain pour la systématiser en règles que l'on code dans un format qui peut permettre à un moteur d'inférence de les utiliser afin d'en déduire les conclusions les plus probables à partir d'un ensemble de faits observés [Lu92].

Il est admis que, quelle que soit l'approche utilisée, meilleures sont les connaissances a priori sur un problème, meilleurs seront les résultats. Pour certains problèmes, nous disposons d'un corps théorique imposant ainsi que de nombreux outils mais pour certains autres, les méthodes classiques peuvent ne mener qu'à des impasses ou des résultats insatisfaisants. Par exemple, si l'on voulait programmer un jeu d'othello champion du monde, l'approche classique demanderait que l'on fouille très profondément l'espace d'état et que nous ayons une fonction heuristique très fiable pour les états sur l'horizon de la fouille, ainsi que des techniques sophistiquées d'élagage. On supposera aussi que le programmeur connaît suffisamment bien le jeu ou qu'il dispose de quelqu'un qui saura transférer ses connaissances expertes dans les fonctions heuristiques. Contrairement à cette approche, nous pourrions laisser le programme construire ses fonctions heuristiques en examinant un très grand nombre de parties [Is93], en apprenant de ses erreurs, et cela aurait ses avantages: non seulement le programme saurait-il jouer à othello mieux que son programmeur mais ce dernier pourrait même obtenir de l'information sur le problème en examinant les fonctions heuristiques générées par son programme.

Les réseaux de neurones artificiels sont un des modèles d'apprentissage populaires. Les réseaux de neurones artificiels sont d'abord apparus avec la cybernétique, dans les années cinquante, où l'on était encore fortement inspiré par l'idée provocatrice de von Neumann selon laquelle l'ordinateur n'était rien de moins qu'un cerveau électronique. La tradition déjà très ancienne des animaux artificiels (jusqu'alors des automates qui, bien qu'étant de pures merveilles d'horlogerie, n'étaient destinés qu'à se répéter indéfiniment) prit un nouvel essort lorsqu'on comprit la versatilité offerte par la programmation. Même si l'on s'est rapidement rendu compte que l'analogie entre l'ordinateur électronique de von Neumann et le cerveau humain n'était qu'une métaphore douteuse, nos notions d'intelligence artificielle sont demeurées fortement influencées par les notions de psychologie et de biologie chères aux cybernéticiens. La recherche en ce sens a donc été dominée par la notion d'*adaptabilité*, propriété qui jusque alors n'avait été observée que dans le monde naturel. Nul ne peut être alors surpris que l'on chercha à s'inspirer très fortement des modèles biologiques.

La programmation génétique [Go89], par exemple, est une métaphore informatique toute darwinienne sur l'évolution d'une espèce (ensemble de chaînes codées) sous l'égide de la sélection naturelle (une fonction objectif) qui décide quels sont les spécimens les plus aptes à la survie. Des mutations stochastiques permettent de produire de nouveaux individus à partir des individus survivants de la génération précédente jusqu'à ce que l'évolution aboutisse enfin à un individu parfaitement adapté (aux yeux de la fonction objectif). Bien que l'on puisse considérer l'adaptabilité du point de vue de l'espèce, comme dans la programmation génétique, il est aussi possible de la considérer du point de vue d'un seul individu. Alors, plutôt que de modifier une population aléatoirement afin de découvrir des propriétés souhaitables, on va plutôt chercher à enseigner à un seul individu un comportement donné, le laissant

découvrir par lui-même le problème. Essentiellement, ce que l'on va chercher à faire, c'est simuler un « cerveau » très simple qui ne comprend que quelques organes sensoriels et quelques « membres » contrôlés par la sortie des neurones. On considérera l'apprentissage comme complet lorsque les « membres » réagiront correctement aux stimuli.

Que ce soit la programmation génétique ou l'apprentissage par réseau de neurones, il s'agit essentiellement d'un problème d'optimisation. Le réseau de neurones représente une fonction que l'on veut optimiser de façon à minimiser l'erreur. Si nous considérons les méthodes classiques pour réaliser cette optimisation, nous faisons deux constats assez gênants. D'une part, le très grand nombre de paramètres libres rend difficile l'obtention de solutions exactes — sans compter que les fonctions calculées par les réseaux de neurones sont non linéaires. D'autre part, dans un problème d'optimisation classique, nous supposons que nous connaissons parfaitement l'espace d'entrée de la fonction, ce qui n'est généralement pas vrai pour plusieurs problèmes intéressants: qui peut caractériser l'espace de tous les caractères manuscrits? Nous allons donc utiliser des méthodes stochastiques qui, on l'espère, nous permettrons de trouver rapidement des solutions satisfaisantes bien qu'approximatives.

Bien qu'ils s'inspirent de la biologie, les réseaux de neurones ne lui sont toutefois pas très fidèles. Les neurones artificiels, à l'instar de leur contrepartie biologique, répondront d'une façon non linéaire aux stimuli qui leur sont envoyés par les autres neurones. On se limitera ordinairement à des connexions à sens uniques et sans circuits. Bien que les synapses biologiques aient un comportement unidirectionnel, on observe par contre des connexions d'une grande complexité (autoconnexions, connexions cycliques, à délai [Is95], etc.) et un mode d'opération asynchrone (les neurones d'une même structure ne répondent pas tous en même temps) alors que dans une certaine classe de réseaux artificiels on considère que les neurones d'une même couche comme synchrones et indépendants. On parle alors de réseaux à étapes (*feed forward*). Un exemple d'un tel réseau est présenté dans la fig.1.

Malheureusement, l'apprentissage avec les réseaux de neurones artificiels a un très sérieux défaut: ils sont très exigeants en temps de calcul car ils exigent un grand nombre d'opérations en virgule flottante, dont certaines non triviales à calculer efficacement, comme la tangente hyperbolique, par exemple. On peut supposer que pour un réseau de n neurones, on aura dans $O(n^2)$ connexions. Cela implique que le temps d'évaluation et d'apprentissage croît très rapidement lorsque le nombre de neurones croît. Une application peut facilement demander de 5000 à 100000 neurones; le réseau qui nous intéresse en présente 6220. Les réseaux de neurones demandent alors des ordinateurs très puissants ou du matériel spécialisé pour être évalués en temps raisonnable pour une application donnée.

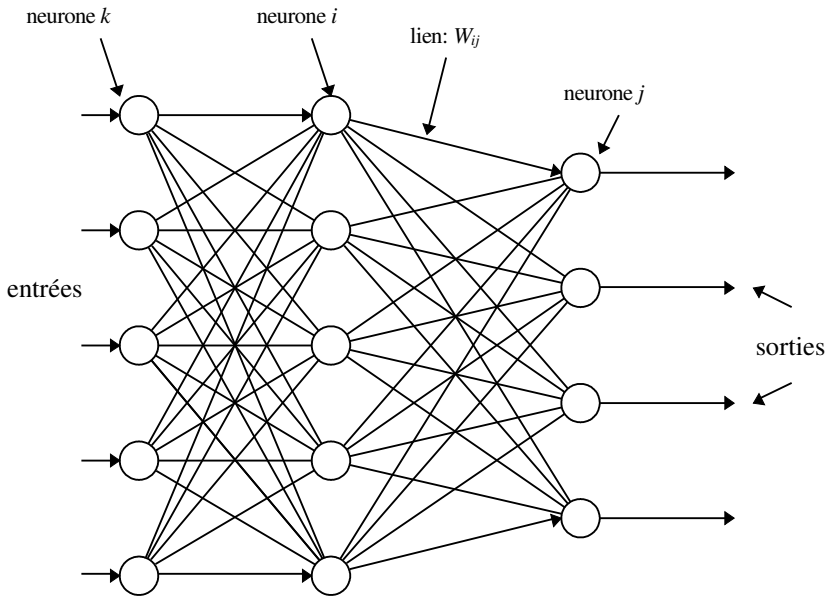


fig. 1. Un réseau de neurones simple. Les neurones à gauche sont des neurones d'entrée, associés aux « organes des sens ». Deux neurones i et j sont reliés par une synapse d'intensité W_{ij}.

1.2 Notation, Évaluation et Rétropropagation.

Nous ne supposerons pas que le lecteur soit totalement étranger au domaine des réseaux de neurones artificiels, mais nous jugeons utile de lui rafraîchir la mémoire et de le familiariser à la notation avant de poursuivre plus avant. Donc, dans cette section, nous présenterons la notation et deux algorithmes utilisés, à savoir l'évaluation d'un réseau à étapes et l'apprentissage par rétropropagation de l'erreur.

1.2.1 Notation.

Le lien entre deux neurones, appelé synapse, contrôle la force de la transmission de l'activation d'un neurone vers l'entrée d'un autre. Cette force, que nous appellerons « poids », est notée w_{ij}, si le lien relie le neurone *i* au neurone *j*. La valeur d'un poids n'est pas contrainte ni par le signe ni par la magnitude. La sortie calculée d'un neurone, dite « activation », est noté x, par opposition à $\hat{x}$, la sortie désirée et idéale. Les activations des neurones des couches cachées (internes) et de sortie sont notés x_i, et les neurones d'entrée du réseau sont notés X_i. Chaque X_i est une activation associée à un vecteur, un exemple, noté simplement X, où $X = \{X_0, X_1, X_2, \ldots, X_n\}$. Les activations sont bornées par les valeurs admissibles de la fonction d'activation, qui est habituellement une fonction sigmoïde. La fonction d'activation d'un neurone *i* notée $\sigma_i(s)$ — car chaque neurone est susceptible d'avoir un certain type, bien qu'en pratique, tous les neurones d'une couche partagent la même fonction d'activation. Il peut s'agir par exemple de la tangente hyperbolique ou une fonction présentant plus ou moins les mêmes caractéristiques. On utilise parfois la fonction logistique, mais la tangente hyperbolique présente l'avantage de pouvoir prendre autant des valeurs négatives que positives, c'est-à-dire d'avoir un effet inhibiteur ainsi qu'excitateur.

1.2.2 Évaluation.

Dans cette section, nous allons présenter dans le détail le calcul les sorties pour le réseau de la fig.1. On peut facilement généraliser à un réseau ayant un nombre quelconque de couches.

Soit *pred(i)*,	la fonction de précédence, qui retourne l'ensemble de tous les neurones *k* tels qu'il y a un lien entre le neurone *k* et le neurone *i*.
Soit *succ(i)*,	la fonction de succession, qui retourne l'ensemble de tous les neurones *j* tels qu'il y a un lien entre le neurone *i* et le neurone *j*.
Soit *sorties*,	l'ensemble des neurones de la dernière couche, qui n'ont pas de successeurs.
Soit $\sigma_i(s_i)$,	la fonction d'activation du neurone *i*.
Soit X_i,	l'activation du neurone *i* s'il s'agit d'un neurone d'entrée.
Soit x_i,	l'activation du neurone *i* s'il ne s'agit pas d'un neurone d'entrée.

L'activation d'un neurone *i* sera donnée par:

$$x_i = \sigma_i(s_i) = \sigma_i(b_i + \sum_{k \in pred(i)} w_{ki} x_k)$$

et où b_i est le biais du neurone *i*. Retenons que

$$s_i = b_i + \sum_{k \in pred(i)} w_{ki} x_k$$

La sortie des neurones de la couche suivante est donc calculée par:

$$x_j = \sigma_j(b_j + \sum_{i \in pred(j)} w_{ij} x_i) = \sigma_j\left(b_j + \sum_{i \in pred(j)} w_{ij} \sigma_i(b_i + \sum_{k \in pred(i)} w_{ki} x_k)\right)$$

Pour le cas particulier des entrées, on peut considérer que la fonction $\sigma_i(X_i)$ est l'identité. On peut par la suite imbriquer indéfiniment les couches, nous n'avons qu'à calculer couche par couche les x_i en prenant soin de respecter la précédence — rappelons que les réseaux à étapes n'ont pas de circuits.

1.2.3 Apprentissage par rétropropagation de l'erreur.

Lorsqu'on présente un exemple au réseau de neurone on obtient une sortie et il est possible qu'elle soit différente de la sortie attendue. Lorsque c'est le cas, on dira qu'il y a une erreur et on voudrait pouvoir la corriger. À cette erreur sera associé un coût, en quelque sorte sa gravité. L'algorithme que nous utiliserons pour modifier les w_{ij} sera la rétropropagation de l'erreur, un algorithme d'optimisation par descente de gradient.

Posons $E_j = \frac{1}{2}(\hat{x}_j - x_j)^2$, la fonction de coût de l'erreur au neurone de sortie *j*, que l'on va chercher à minimiser, et

$E = \sum_{j \in sorties} E_j$, le coût total de l'erreur.

On veut donc ajuster les paramètres du réseau, les w_{ij}. Nous allons pour cela calculer les dérivées de l'erreur par rapport aux poids. Nous les modifierons ensuite en utilisant la descente de gradient. En particulier:

$\frac{\partial E}{\partial x_j} = -(\hat{x}_j - x_j)$, la dérivée par rapport à la sortie *j* du réseau. Nous avons aussi que

$$\frac{\partial E}{\partial w_{ij}} = -(\hat{x}_j - x_j)\frac{\partial x_j}{\partial w_{ij}} = -(\hat{x}_j - x_j)\frac{\partial \sigma_j(s_j)}{\partial w_{ij}} = -(\hat{x}_j - x_j)\frac{\partial \sigma_j(s_j)}{\partial s_j}\frac{\partial s_j}{\partial w_{ij}}$$

qui est la dérivée de l'erreur par rapport au poids qui relie un neurone caché *i* au neurone de sortie *j*. Le gradient de l'erreur au neurone *i* est:

$$g_i = \frac{\partial E}{\partial x_i}\frac{\partial x_i}{\partial s_i} = \frac{\partial E}{\partial s_i}$$

mais pour un neurone de sortie *j* nous avons:

$$g_j = \frac{\partial E_j}{\partial x_j}\frac{\partial x_j}{\partial s_j} = -(\hat{x}_j - x_j)\sigma'_j(s_j)$$

alors que pour un neurone caché *i* nous avons:

$$g_i = \frac{\partial E}{\partial s_i} = \frac{\partial E}{\partial x_i}\frac{\partial x_i}{\partial s_i} = \sigma'_i(s_i)\sum_{j \in succ(i)} w_{ij} g_j$$

Nous calculerons les modifications sur les poids à l'aide d'un paramètre supplémentaire, η, que l'on nomme le pas de gradient. Pour les couches où l'on de dispose pas des sorties désirées (les $\hat{x}$ idéaux), nous calculerons la dérivée de l'erreur par rapport à un poids avec le gradient de l'erreur plutôt qu'avec l'erreur elle-même. L'on obtient donc que:

$$\Delta w_{ij} \quad = \quad -\eta \frac{\partial E}{\partial w_{ij}} \quad = \quad -\eta \frac{\partial E}{\partial s_j}\frac{\partial s_j}{\partial w_{ij}} \quad = \quad -\eta\, g_j x_i$$

et finalement le nouveau poids sera $w'_{ij} = w_{ij} + \Delta w_{ij}$

Et nous disposons déjà des g_j, puisque calculé à la couche supérieure. Prenons note que nous ne calculerons pas les gradients associés aux neurones d'entrée, car nous ne faisons pas de rétropropagation passé ce point. Il serait en effet inutile de calculer ces gradients lorsque nous n'avons pas de poids à ajuster avant les entrées.

Les biais, dont nous avons parlé pour la passe avant, peuvent être calculés par les équations ci-dessus en supposant qu'il existe un neurone x_b caché dont la valeur est toujours 1 ou -1, qui est relié au neurone *i* par le poids b_i. L'activation du neurone x_b peut être -1 ou 1, indifféremment, car l'apprentissage ajustera le biais en conséquence. Pour plus de détail sur la rétropropagation, on peut consulter [He91] ou [Me92], ou sur les réseaux de neurones en général on peut consulter [Ha94].

1.2.4 Convergence, capacité et qualité.

L'apprentissage peut être caractérisé selon deux critères importants. Le premier critère, la convergence, détermine si le réseau apprend et si sa performance se stabilise. Si le réseau converge, nous observerons que la performance (quelque en soit la mesure), au long de l'apprentissage, rejoint une valeur limite. Si la performance oscille sans rejoindre une valeur limite (ou rejoint une valeur triviale) nous parlerons de *divergence*.

La valeur limite de la performance détermine la *qualité* de l'apprentissage. Bien que la *qualité*, pour être correctement évaluée, demande que l'on tienne compte du nombre de paramètres libres, de l'architecture et du nombre d'exemples présentés au réseau, elle est une bonne indication de la justesse des choix architecturaux du réseau. Si nous constatons que la qualité atteinte est insuffisante, nous pouvons penser à d'autres architectures. La qualité peut aussi servir à comparer diverses solutions pour un même problème.

La capacité est une mesure plus difficile à obtenir et est un bon indicateur sur les qualités intrinsèques d'un réseau. Pour un problème de classification, nous pouvons simplifier les choses en disant que la capacité d'un réseau de neurones, c'est le nombre d'exemples qu'il peut apprendre « par cœur ». C'est à dire que la capacité est la taille maximale de la population d'exemples pour laquelle le réseau donne *toujours* la bonne réponse. Plus on dépasse la capacité, plus un réseau est susceptible de se tromper.

1.3 Réseaux LeNet 1a, convolutions et parallélisme.

Dans la plupart des réseaux de neurones appliqués à des problèmes spatiaux (reconnaissance de caractères, détection de formes, analyse d'une position d'othello, etc.) on utilise des sous-structures nommées *feature maps*, ou *cartes*, que l'on conçoit selon nos connaissances a priori sur le problème pour faire converger plus rapidement l'apprentissage. Par exemple, nous avons conçu différentes cartes en fonction du rôle qu'elles devaient jouer dans l'analyse d'une position à othello. Certaines cartes donnaient une grande importance aux coins, d'autres aux côtés et autres régions. Nous avons démontré que la conception soignée de cartes pouvait augmenter significativement la performance d'un réseau (dans notre cas, modifier une seule carte mena à une augmentation de 10% dans l'exactitude des prédictions!).

En utilisant sa connaissance a priori Le Cun [Le90], [Sä92] conçut une foule de réseaux utilisant les cartes. Ces réseaux, nommés LeNet, sont divisés en deux parties distinctes. La première partie de ces réseaux (dont nous avons choisi le LeNet 1a comme problème-type) est justement un assemblage de cartes où la connectivité des neurones d'une carte se réduit à une convolution en deux dimensions. Ce type d'opération est connu depuis longtemps en traitement de signaux classique (son et image) et ses propriétés ont été très attentivement étudiées (consulter par exemple [Ly94]). C'est justement cette idée que Le Cun exploite. Après tout, la reconnaissance de caractère, c'est essentiellement de la reconnaissance d'image. Il est donc possible de concevoir des cartes qui mettent en évidence certaines caractéristiques de l'image, comme les courbes, les obliques, etc., et de construire un réseau de neurones classique qui puisse prendre en entrée les sorties de la passe « traitement d'image ».

C'est ce qui se trouve dans le réseau LeNet 1a (voir l'annexe 1 pour tous les détails). Seulement, contrairement à une application « traitement d'image » où les filtres (dont les coefficients forment les *noyaux*) sont entièrement connus d'avance ou ajustés manuellement, le réseau LeNet permet d'effectuer l'apprentissage sur les noyaux. Comme on applique un seul noyau par carte, on partage les w_{ij}, les coefficients du noyau, réduisant ainsi le nombre de paramètres libres, ce qui a pour effet de restraindre le domaine de recherche. Cette caractéristique permet d'obtenir des filtres plus efficaces et mieux adaptés que ceux qu'aurait pu être conçus par le chercheur.

Dans le réseau LeNet 1a, la reconnaissance est réalisée grâce à quatre couches de type convolution et une seule de type pleinement connectée, où tous les neurones de la couche précédente sont reliés à tous les neurones de la couche suivante. Globalement, le réseau LeNet prend en entrée une image numérique de 32x32 pixel (voir fig.2 de l'annexe 1), la décompose en plusieurs cartes, lesquelles sont à

leur tour décomposées, pour être finalement recombinées dans la couche pleinement connectée qui a 10 sorties. La sortie la plus excitée identifie le chiffre reconnu.

Les convolutions forment la plus grande part des calculs dans un réseau de ce type et on observe que l'opération de convolution sur une image demande beaucoup d'opérations, soit $O(k^2(n-k+1)^2)$, où k^2 est la taille du noyau, et n^2 la taille d'image.

Une solution intéressante pourrait être d'assigner un processeur par série de cartes (de façon à ce que seules les cartes dépendantes les unes des autres se trouvent sur un même processeur) ce qui a priori semble bien convenir aux machines faiblement parallèles qui sont maintenant assez courantes [Sä92]. Bien que ce modèle soit très intéressant pour le multitâche où des processus indépendants sont exécutés sur différents processeurs, l'efficacité diminue grandement lorsqu'on fait communiquer un grand nombre de sous processus entre eux par message ou par mémoire partagée. Bien que l'on obtienne globalement un gain, la connectivité parfois très élevée des cartes peut rendre compliquée la gestion de la communication des valeurs partagées entre les différentes cartes, ce qui mène à des performances moins élevées. Mais cette solution qui augmente grandement la performance exploite quand même moins de parallélisme qu'il est possible d'exploiter.

La convolution a en effet un potentiel de parallélisme très élevé. Certains microprocesseurs, par exemple le Pentium d'Intel, sont capables d'exploiter le *parallélisme au niveau de l'instruction*, ou ILP. Cependant, pour un programme quelconque, il est admis que le parallélisme au niveau de l'instruction exploitable est habituellement de l'ordre 6 à 7 instructions [Ka94]. Cette amélioration, bien qu'elle permette un gain important, nous laisse encore avec un calcul en $O(n^2)$, ce qui est très loin du potentiel de parallélisme de la convolution. Si nous exploitons pleinement le parallélisme de la convolution, nous obtiendrons un algorithme en $O(\log k)$. Comme il est matériellement difficile (sinon impossible) de réaliser un circuit pour cet algorithme nous chercherons plutôt à profiter d'un parallélisme du même ordre que le nombre de pixels, soit en $O(k^2)$.

Considérons l'option du parallélisme massif, où nous assignons un processeur par neurone. Il existe des ordinateurs avec un nombre suffisamment élevé de processeurs, mais ces processeurs sont habituellement des processeurs généraux ce qui, pour des réseaux de neurones, est du gaspillage de matériel puisque seulement quelques opérations sont nécessaires: la multiplication, l'addition, la fonction sigmoïde pour l'évaluation, et la dérivée pour la rétropropagation de l'erreur lors de l'apprentissage. De plus, dans les réseaux de neurones pleinement connectés, la distribution de l'information devient un très sérieux problème. Dans les machines parallèles actuelles possédant un aussi grand nombre de processeurs, les réseaux de communication ne sont pas toujours très efficaces pour les communications globales. C'est qu'un neurone peut être branché à plusieurs autres; si chacun de ces neurones se trouve sur un processeur

différent, il faut bien trouver le moyen de communiquer le plus efficacement possible la valeur d'un neurone vers toutes les destinations. Nous n'entrerons pas plus dans les détails concernant les architectures des multiprocesseurs contemporains (consulter [Ka94] qui en discute de façon exhaustive) mais nous verrons que dans le cas du LeNet un compromis intéressant est possible.

Nous devons ignorer les multiprocesseurs courants parce qu'ils ne sont pas suffisamment adaptés ou trop dispendieux, et que, par conséquent, ils ne conviennent nullement aux applications pratiques! Il est bien évident que si nous voulions fabriquer une machine qui doit lire les codes postaux sur les enveloppes, par exemple, devoir la brancher à un superordinateur n'est ni économique ni pratique. Ce qu'il nous faut, c'est une machine d'un coût abordable et de dimension suffisamment petite pour être intégrée à une autre machine, par exemple un télécopieur. Si le prix de ce télécopieur amélioré demeure comparable au prix d'un télécopieur normal (soit environ de 500$ à 1000$) le produit trouvera son acheteur. S'il est beaucoup plus dispendieux, personne n'en voudra, même si les avantages sont importants.

Il nous faut alors chercher dans une autre direction: celle des machines dédiées qui, généralement, peuvent être produites en grand nombre et à faible coût. Parmi ces machines dédiées, on trouve les processeurs de traitement de signaux numériques (ou DSP), les modems et les contrôleurs d'entrées/sorties. Ces puces sont disponibles à peu de frais (une vingtaine de dollars pour un DSP, dix pour un modem) et en grand nombre au besoin.

1.4 Matériel spécialisé et calcul.

Lorsque nous envisageons la construction d'une machine dédiée, nous pouvons nous permettre des optimisations que nous ne pourrions effectuer s'il s'agissait d'un processeur général. Nous devons concentrer nos efforts sur l'optimisation du matériel afin de minimiser le temps de calcul, et élaborer le matériel de façon à ce qu'aucune fonction superflue ne soit incluse. Dans notre cas, ce que nous voulons, c'est réaliser les convolutions le plus rapidement possible. Nous savons qu'il est possible de réaliser la multiplication scalaire de deux matrices $n \times n$ en $O(n)$ avec un algorithme parallèle [Qu89]. L'algorithme que décrivent Quinton et Robert s'exécute sur une machine parallèle dont le modèle est connu sous le nom de processeur systolique. Kai Hwang ([Ka94], p. 84) résume ainsi les principales caractéristiques des processeurs systoliques:

> This is a class of multidimensional pipelined array architecture designed for implementing fixed algorithms [...] In general, static systolic arrays are pipelined with multidirectional flow of data

> streams. [...] With fixed interconnections and synchronous operation, a systolic array matches the communication structure of the algorithm. For special application like signal/image processing, systolic arrays may offer a better cost/performance ratio.

Nous discuterons en détail dans un chapitre ultérieur de l'architecture systolique ainsi que des algorithmes parallèles que nous utiliserons. Pour l'instant, il est important de concevoir la dichotomie entre un processeur *dédié* et un processeur *général*.

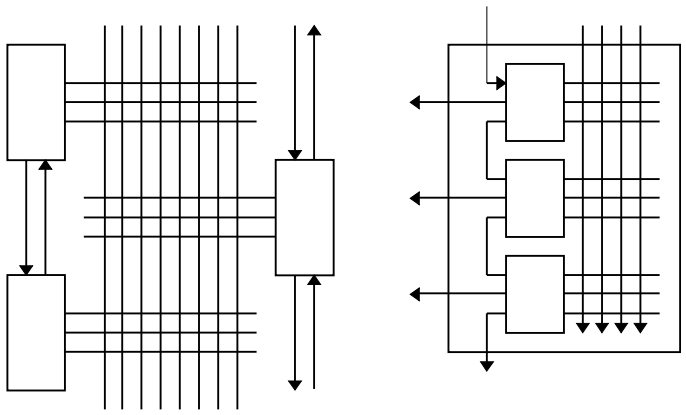
La méthode traditionnelle, lorsqu'on débute la conception d'un microprocesseur dédié, implique que l'on s'enferme pendant au moins un an, au terme d'une période d'étude, dans son laboratoire afin de le dessiner transistor par transistor [Co94]. On imbrique par la suite le microprocesseur dans un système asservi (une carte d'extension, par exemple) qui prend aussi un certain temps à développer. Bien que cette approche mène à des performances très intéressantes, elle a le très sérieux désavantage d'être vulnérable (et de façon définitive) aux décisions prises au début de la conception. Le coût prohibitif du développement d'une telle puce (par exemple, environ deux cent mille dollars américains pour la puce NET32K d'AT&T dédiée au traitement d'image) rend quasi impossibles l'abandon du produit ou encore une reconception complète. Si le produit est très bien, c'est tant mieux, mais s'il s'avère que la performance n'est pas celle escomptée, c'est bien tant pis!

Par le passé, développer les processeur en VLSI était la seule solution disponible, mais de nouvelles technologies offrent maintenant des alternatives intéressantes. Que l'on parle de FPGA, EPLD ou autres, la caractéristique principale de ces circuits à logique programmable est de pouvoir simuler en matériel un circuit logique à partir d'une description produite par un compilateur et un langage de haut niveau, comme VHDL ou des variantes comme AHDL [Al95]. De façon très simplifiée, une telle puce consiste en un ensemble de cellules logiques (une porte logique universelle programmable munie d'un bit de mémoire) et d'un réseau d'interconnexions global. Les circuits à logique programmable supportent aussi des vitesses d'horloges assez élevées (dépendant du chemin critique du circuit simulé) allant de 1 à 100Mhz, ce qui est comparable aux bons microprocesseurs du début des années '90. Il faut aussi préciser que les circuits à logique programmable n'offrent généralement pas de portes analogiques, ce qui nous force à exclure des solutions intéressantes comme [Co94].

La caractéristique la plus intéressante de plusieurs circuits à logique programmable est qu'ils sont configurables à la volée. Ce qui signifie qu'une carte d'extension (par exemple la V.I.P. [Cl95]) peut être dessinée de façon à permettre une programmation par l'hôte (un PC/Pentium dans notre cas) à n'importe quel moment. De plus, les temps de configuration sont en général assez intéressants pour pouvoir considérer, selon les applications, de reprogrammer les circuits entre les étapes du calcul ou entre les

applications. Donc, en théorie, une carte pourrait contenir un certain nombre de circuits à logique programmable qui pourraient être reprogrammés selon la tâche à accomplir, formant ainsi un coprocesseur toujours très efficace pour une tâche donnée.

L'inconvénient majeur des circuits à logique programmable est leur relativement faible capacité en terme du nombre de portes logiques. Bien que les constructeurs offrent des modèles dont on peut *espérer* utiliser de 20000 à 30000 portes logiques universelles, la réalité du routage fait qu'en pratique seulement une fraction de ces portes sont utilisables (considérer la fig.2). Les différents constructeurs de circuits à logique programmable offrent différents modèles de connexion qui peuvent rendre difficile la conception d'un circuit. Altera, par exemple, n'offre que des connexions globales (à l'extérieur d'un groupe de cellules logiques) ce qui consomme très rapidement les ressources, tandis que les circuits de Xilinx, qui permettent des connexions très complexes, locales et segmentées, rendent, eux, la prévision des délais très difficile! De plus, les compilateurs ne procèdent jamais au routage par un algorithme exact (car on sait que cela appartient à la classe des problèmes NP-complets [Ku90]) mais par une heuristique qui est susceptible de ne pas utiliser tout le matériel disponible ou de ne pas l'utiliser de façon optimale.



(a) (b)

fig. 2. Communication dans un circuit à logique programmable Altera. (a) Les blocs et leurs branchements sur le réseau d'interconnexions global. Tous les blocs ne communiquent pas directement entre eux, seuls les voisins immédiats profitent d'une connexion privilégiée. (b) Détail d'un bloc. Plusieurs cellules logiques sont encapsulées dans une même unité. Cela leur permet de partager un réseau local d'interconnexions et aussi d'échanger rapidement de l'information via la liaison de retenue.

Il est donc difficile d'implanter des circuits logiques qui s'étendent sur plusieurs blocs (un groupe de cellules logiques). Ce qui nous force donc à ne choisir que des opérations relativement simples et demandant très peu de portes logiques. Heureusement, comme nous verrons au chapitre suivant, il existe des opérations très simples à réaliser en matériel qui sont à peu près équivalentes (pour un réseau de neurones artificiel, du moins) à la multiplication-accumulation en nombres à virgule flottante, qui tiennent sur moins de blocs et consomment moins de connexions.

Donc, nous supposons qu'un matériel hautement spécialisé, comme par exemple un processeur systolique à deux dimensions et un système arithmétique à faible précision, peut mener à des performances très intéressantes — sous l'hypothèse que l'efficacité des réseaux de neurones demeure comparable. Nous pouvons résumer les avantages espérés ainsi: un très haut degré de parallélisme, dans l'ordre du nombre de pixels (ce qui correspond au nombre de neurones dans le cas des cartes), et une très grande simplicité des opérations. De plus, comme avantage supplémentaire, on peut noter la reconfigurabilité du matériel à la volée, garantissant ainsi que le matériel est toujours le plus efficace possible pour la tâche donnée. Les désavantages sont qu'à cause du petit nombre de portes logiques universelles et de la (relativement) faible connectivité des circuits à logique programmable, les processeurs doivent rester de taille modeste donc arbitrairement simples. Chaque nouvelle configuration nécessite aussi une bonne partie de conception matérielle, opération peu évidente pour les programmeurs qui n'ont peu ou pas d'expérience avec le niveau matériel de l'informatique.

Chapitre 2: Calcul à précision réduite et réseaux de neurones artificiels.

On a proposé [Cl94], [Si94] d'utiliser des poids de faible précision et des valeurs d'activations qui ne s'expriment qu'en des puissances négatives de deux. Nous verrons qu'en ce cas bien particulier, nous pouvons remplacer la multiplication par un décalage, une opération beaucoup plus facile à réaliser en matériel. Nous allons nous attarder dans ce chapitre aux moyens à mettre en œuvre pour réaliser ces transformations en logiciel comme en matériel. Nous insisterons autant sur un aspect que sur l'autre, car, comme nous le verrons plus tard, réaliser les algorithmes d'apprentissage en matériel pose encore des problèmes, et l'apprentissage en logiciel est préférable parce que plus souple.

2.1 Nouvelles opérations et notation.

Nous allons utiliser des nombres à point fixe plutôt que des nombres à virgule flottante afin de simplifier grandement le matériel. Pour représenter une fraction à l'aide d'un entier en complément à deux, nous choisissons un dénominateur, m, et nous exprimerons le nombre en m-ièmes. Cela permet, notamment, de procéder directement à une addition car le résultat demeure valide en m-ièmes. Le dénominateur m est en relation directe avec le nombre de bits que nous voulons utiliser. Si nous voulons utiliser un entier de n bits, dont d bits pour les décimales, nous aurons un dénominateur $m = 2^d$. Nous aurons alors n-d-1 bits pour représenter la partie entière du nombre — car il ne faut pas oublier que nous avons aussi un bit de signe.

Pour ce qui est des activations, nous utiliserons des nombres de la forme $(-1)^s 2^{-e}$. Nous aurons un bit de signe, *s*, et quelques bits pour l'exposant, *e*. Selon la précision de l'exposant, nous pouvons représenter les nombres de la fig. 3. Nous devons surtout voir que plus nous avons de bits, plus nous pouvons représenter une petite quantité, ce qui n'a pour effet que d'augmenter la résolution autour de 0. Nous verrons qu'il paraît suffisant de n'avoir que 3 bits pour une activation, incluant le bit de signe. Avec 3 bits, nous pouvons représenter $\pm 1/8$ comme nombres les plus près de 0.

		Valeurs admissibles
Logiciel	Poids (*n* bits)	$\{-2^{n-1}/m,\ldots,0,\ldots,(2^{n-1}-1)/m\}$
	Activations (3 bits)	$\{-1,-1/2,-1/4,-1/8,1/8,1/4,1/2,1\}$
Matériel	Poids (*n* bits)	$\{-2^{n-1}/m,\ldots,0,\ldots(2^{n-1}-1)/m\}$
	Activations (3 bits)	$\{-1,-1/2,-1/4,-1/8,1/8,1/4,1/2,1\}$

Fig. 3. Valeurs admissibles pour les différents types.

Pour les valeurs représentables, nous devons séparer les poids et les activations. Les poids en point fixe seront notés w_{ij} et déterminés par le contexte, alors que les activations seront notées $\tilde{x}_i$. Ainsi, le lecteur devra comprendre que $w_{ij}\tilde{x}_i$ est un produit à faible précision alors que $w_{ij}x_i$ sera un produit de précision arbitraire. La fonction sigmoïde, $\tilde{\sigma}_j(\tilde{s}_j)$, que nous utiliserons est une fonction linéaire par partie, définie sur tout le domaine, soit:

$$\tilde{s}_j = \sum_{i \in pred(j)} w_{ij}\tilde{x}_i$$

$$\tilde{\sigma}_j(\tilde{s}_j) = \begin{cases} 1, & si \quad \tilde{s}_j \geq \frac{3}{4} \\ \frac{1}{2} & si \quad \tilde{s}_j \geq \frac{3}{8} \\ \frac{1}{4} & si \quad \tilde{s}_j \geq \frac{3}{16} \\ \frac{1}{8} & si \quad \tilde{s}_j \geq 0 \\ -\frac{1}{8} & si \quad \tilde{s}_j \geq -\frac{3}{16} \\ -\frac{1}{4} & si \quad \tilde{s}_j \geq -\frac{3}{8} \\ -\frac{1}{2} & si \quad \tilde{s}_j \geq -\frac{3}{4} \\ -1 & autrement \end{cases}$$

$$\frac{\partial \tilde{\sigma}_j(\tilde{s}_j)}{\partial \tilde{s}_j} = \begin{cases} 1 & si \quad \tilde{s}_j \in]-1,1[\\ \frac{1}{32} & autrement \end{cases}$$

La valeur de la dérivée à l'extérieur de l'intervalle]-1,1[dépend de la plus petite activation représentable en matériel, si le matériel peut supporter l'apprentissage. Autrement la valeur doit être assez petite, comme par exemple $^1/_{32}$, ce que nous utilisons pour l'apprentissage en logiciel. Introduisons aussi $\delta(g)$, la fonction de discrétisation du gradient, telle que:

$$\tilde{g} = \delta(g) = signe(g)\, 2^{\left[\log_2(|g|)\right]}$$

$$signe(g) = \begin{cases} +1 & si \quad g \geq 0 \\ -1 & autrement \end{cases}$$

Et où $\lceil x \rfloor$ est l'entier le plus près de x et $|x|$ la valeur absolue de x. Lors de l'apprentissage, il pourra être intéressant d'exprimer aussi les gradients en termes de puissances de deux. Le calcul du gradient deviendrait alors

$$g_i = \tilde{\sigma}_i'(\tilde{s}_i) \sum_{j \in succ(i)} w_{ij} \delta(g_j) = \tilde{\sigma}_i'(\tilde{s}_i) \sum_{j \in succ(i)} w_{ij} \tilde{g}_j$$

Cette façon de calculer le gradient, si elle est équivalente à la formule présentée au chapitre précédent, permettra de n'utiliser qu'un décaleur pour faire la multiplication. Nous avons dit brièvement qu'elle se réduisait à un décalage lorsqu'on considère que les activations ne peuvent prendre que des valeurs qui sont des puissances de deux. C'est qu'en base deux, multiplier ou diviser par deux revient à déplacer le point vers la droite ou la gauche. Nous remarquons qu'avec les gradients, nous avons des puissances positives *et* négatives, mais qu'avec les activations nous n'avons que des puissances *négatives* de deux. Cela ne permet que la division, qui est, dans ce cas, un décalage du point vers la gauche. Comme nous utilisons des nombres à point fixe, plutôt que de déplacer le point, nous allons décaler le nombre vers la droite, sans contrôle d'arrondi.

Lorsque nous n'avons que des décalages vers la droite (avec possiblement un décalage de 'zéro' chiffres), cela présente l'avantage de ne demander qu'un circuit très simple en matériel. Nous observons qu'en logiciel, toutefois, que le rapport d'efficacité entre les multiplications entières et les décalages dépend fortement du processeur sur lequel on exécute le programme. Le Pentium permet la multiplication pipelinée, réalisant ainsi une multiplication à tous les cycles, alors que le 80486 met de 13 à 26 cycles pour une multiplication d'entiers tandis qu'il prend trois cycles pour un décalage de *n* bits et un seul pour un masque [Hu92]. Le résultat de la multiplication est un nombre en point fixe en complément à deux qu'il nous reste à additionner à l'aide de l'opération d'addition habituelle.

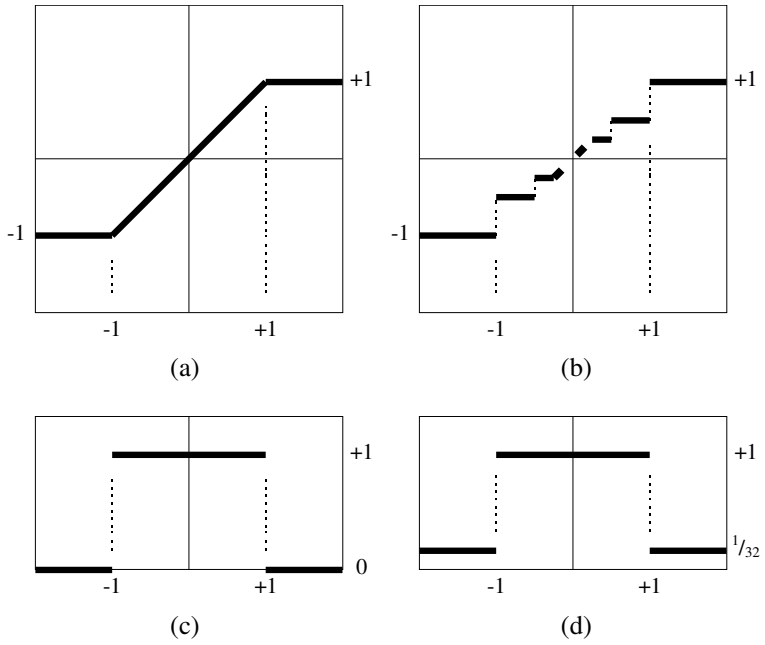


Fig. 4. (a) une fonction linéaire par partie et (b) son approximation en puissances négatives de deux. (c) La dérivée de (a) a aussi sa contrepartie, en (d) la dérivée (b) exprimée en puissances négatives de deux.

2.2 Impacts sur la précision des poids et des activations.

Il est généralement admis que la précision minimale pour les poids est de 16 bits alors que 8 bits seraient suffisants pour la passe avant [Cl94,Si94,Cl96,Ki96,Ly96]. En effet, les tests que nous avons faits semblent démontrer que 16 bits sont nécessaires pour que l'apprentissage converge et que moins de bits rendent soit l'apprentissage trop lent ou encore totalement impossible. Des tests faits avec aussi peu que 12 bits démontrent que le réseau n'apprend pas du tout, quel que soit le pas de gradient, η. Pour l'évaluation simple, il nous semble que 12 bits soient un minimum, bien que des résultats intéressants aient été obtenus avec 8 bits [Ly96]. Pour ce qui est des activations, il a été spéculé que de très petites quantités devraient pouvoir être représentées, par exemple par un exposant sur 5 bits. L'exposant est présentement représenté par un nombre variable de bits, mais il semblerait que 3 bits d'exposants soient suffisants.

Cependant, nous devons nous demander quelle est la quantité d'erreur que nous introduisons en discrétisant fortement les poids et les activations. Lorsque nous utilisons un `float`, nous disposons d'un bit de signe, de huit bits d'exposant et enfin de 24 bits de mantisse (dont un virtuel), et la plus petite quantité représentable peut être extrêmement petite (dans l'ordre de 10^{-38} pour un `float`). Cependant, les exposants des nombres doivent être assez près les uns des autres pour que les opérations se réalisent sans

grande perte de précision. Dans le cas des réseaux de neurones, nous semblons effectivement être confinés aux 24 premiers bits [Ly96, Ki96]. Ce qui signifierait que 0,00000006, ou 2^{-24}, serait assez près de la dernière quantité significative utilisée. Par contre, si nous utilisons 15 bits significatifs (encore, on a exclu le bit de signe), la plus petite quantité représentable est donc 2^{-15}, soit environ 0,00003, ce qui est bien plus grand que la valeur trouvée pour les `float`! Bien que cela puisse sembler une assez grande différence relative, le pas de gradient est plutôt dans l'ordre de 0,01 à 0,001, ce qui est très largement supérieur à la plus petite quantité représentable par l'un ou l'autre.

Il est évident que si l'on réduit le nombre de bits, la précision des quantités représentées sera moindre; cependant, le lecteur pourra consulter [Cl94] pour une analyse détaillée de l'introduction du bruit liée à la discrétisation et de la compensation qui devrait être appliquée au paramètre η pour que l'apprentissage congerve encore assez rapidement. Intuitivement, nous pouvons dire que si les modifications sur les poids ne touchent pas au moins les 2 ou 3 bits du bas, il faut augmenter le pas de gradient η, du moins lors des premières époques. Une époque est un cycle d'apprentissage pendant lequel nous présentons au réseau tous les exemples. Nous présumons cependant que l'impact se ressentira principalement au niveau de la capacité du réseau. La capacité représente le nombre d'exemples que le réseau sera capable d'apprendre « par cœur ». À un nombre de paramètres égal, un réseau dont les poids sont de faible précision a une capacité grandement réduite par rapport à un réseau dont les paramètres sont représentés avec une plus grande précision. Nous verrons au chapitre 5, où nous présenterons nos résultats quant à l'apprentissage, comment la réduction de la capacité diminue la performance d'un réseau de neurones. Nous y discuterons aussi des moyens à prendre pour remédier à cet effet.

2.3 Apprentissage.

Nous démarrons l'apprentissage avec des poids initialisés au hasard, selon une loi uniforme sur l'intervalle prescrit par le dénominateur *m*, c'est-à-dire]-*m*, *m* [et de moyenne 0. Les premières époques servent à faire les ajustement grossiers, alors on doit utiliser un η assez grand [Cl94]. Mais plus les époques passent, plus η devrait être petit pour permettre les ajustement fins car l'erreur E descend très rapidement et un pas de gradient trop élevé mène à l'instabilité.

L'apprentissage par l'algorithme de rétropropagation change très peu. Rappelons que la caractéristique principale de la sigmoïde discrète est qu'elle est linéaire par partie et que sa dérivée a deux points de discontinuité alors que la tangente hyperbolique est lisse en tout point, comme sa dérivée. Ces différences peuvent être cependant balancées par un pas de gradient adéquat.

L'algorithme d'apprentissage par rétropropagation de l'erreur consiste essentiellement à explorer la surface de la fonction d'erreur dans un espace à n dimensions (où n est le nombre de poids ou paramètres) afin de trouver un minimum. Ce qui change le plus dans l'algorithme d'apprentissage lorsque nous utilisons une faible précision, c'est la façon dont nous explorons l'espace des paramètres. Nous trouvons un vecteur en n dimensions mais dont la faible précision ne permet pas d'indiquer très précisément la direction de la descente. Cela a pour effet de rendre plus difficile la descente dans les minima locaux et le gradient peut très facilement nous en faire sortir si le pas de gradient, η, est trop grand, ou encore on y reste s'il est trop petit. Le choix du paramètre η est d'autant plus important que la précision est faible. Nous discuterons de ces effets au chapitre 5, où nous présentons les résultats.

Chapitre 3: Algorithmes de convolutions rapides

Avant de penser à remplacer le matériel présentement disponible par un matériel mieux adapté et plus performant, il est raisonnable de se demander s'il n'y a pas quelque chose à faire pour accélérer le traitement sur un ordinateur général. Bien entendu, nous pouvons utiliser toutes les options des compilateurs optimisants offerts par les constructeurs des différents postes de travail mais nous pouvons montrer assez facilement que les optimisations réalisées — bien qu'intéressantes — sont insuffisantes parce qu'elles n'exploitent que le parallélisme au niveau de l'instruction offert par le processeur, ce qui est très en deçà du parallélisme exploitable. Avant d'explorer les solutions à ce problème, définissons d'abord en détail ce qu'est une convolution.

3.1 Qu'est-ce qu'une convolution?

Nous avons brièvement introduit la notion de convolution, sans toutefois expliquer en détail comment cette opération pouvait être appliquée sur une image. Donc, avant de poursuivre, nous allons décrire très exactement ce que nous entendons par une convolution, et ce que ça signifie que d'appliquer une convolution sur une image. La convolution est un produit scalaire de deux matrices, l'une étant un segment de l'image, l'autre étant le noyau. Supposons, sans perte de généralité, que l'image et le noyau soient carrés. Soient $k \times k$, la taille du noyau et K_{ij}, ses coefficients. Soient $n \times n$, la taille de l'image et $x_{a,b}$, les pixels. Alors la convolution appliquée au tour du pixel $x_{a,b}$, tels que a et b sont inférieurs à $n-k+1$ est définie par l'opération suivante:

$$c_{a,b} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} K_{ij} x_{i+a,j+b}$$

Optionnellement, nous pouvons appliquer une fonction telle que $c'_{a,b} = f(c_{a,b})$. Dans le cas qui nous intéresse, il s'agira de la sigmoïde, mais il pourrait s'agir d'une fonction quelconque. Le résultat, l'image de tous les $c_{a,b}$, est plus petite que l'original, car pour une une région de k-1 pixels de large, la somme $i+a$ ou $j+b$ dépasse n, et indexe un pixel hors de l'image originale. La taille de l'image finale sera donc de $(n-k+1)^2$ pixels.

3.2 Convolutions rapides en logiciel pour un ordinateur général.

L'opération de convolution demande dans $O(k^2(n-k+1)^2)$ opérations. Puisque cela demande beaucoup d'opérations dès que n ou k devient grand, nous voudrions disposer de moyens efficaces de réaliser la convolution. Nous verrons que, dans certains cas restreints, il est possible d'atteindre des performances intéressantes mais que ces méthodes s'appliquent plutôt mal aux convolutions que l'on retrouve dans les réseaux de neurones.

Pour réaliser un programme qui calcule efficacement la convolution, nous pouvons nous en remettre aux techniques habituelles et au compilateur. Par exemple « dérouler » les boucles mène à une augmentation de la performance, mais cela suppose que l'on écrive du code très spécifique. Plutôt que de représenter les boucles avec les primitives habituelles, on va faire l'expansion de tous les termes. Cela minimise la gestion des données pendant l'exécution du programme, en plus de se sauver les branchements conditionnels qui réduisent quelque peu la performance [Ka94].

On peut cependant augmenter substantiellement la performance en créant des algorithmes de « convolution rapide » qui dépendent très fortement du noyau. Comme pour la transformée de Fourrier, par exemple, où beaucoup de termes se répètent, nous obtenons une *transformée rapide* en exploitant au maximum la redondance dans les produits et les sommes partielles. L'application d'un noyau symétrique ou avec assez peu de coefficients différents de zéro sur une image peut être codée efficacement si on peut se permettre de générer du code sans l'intervention de l'utilisateur.

Dans le cas où nous pouvons nous permettre d'avoir un petit nombre de noyaux, tous prédéfinis et statiques, la génération de code source est très pratique. Il s'agit seulement d'élaborer un protocole qui puisse permettre de chaîner facilement les procédures générées au programme principal. Dès que cela est fait, écrire le programme qui prend un noyau et une taille d'image en entrée et génère le code C/C++ en sortie est facile à faire, car il s'agit d'une variation sur le thème de l'élimination des sous-expressions

communes [Ah86]. Comme ce code n'a pas la contrainte d'être lisible par le programmeur, on peut inclure toutes sortes d'optimisations, comme dérouler complètement les boucles, travailler avec des pointeurs plutôt que des tableaux, créer des variables temporaires pour éliminer les calculs redondants, placer côte à côte les instructions qui partagent les mêmes constantes (pour profiter du fait qu'une constante demeure chargée dans un registre tant qu'elle est utilisée ou doit être remplacée par une autre), etc. La génération finale du code et les optimisations dépendantes de la machine sont gérées par le compilateur. Évidemment, le désavantage majeur de cette technique est que le code généré est statique: modifier un noyau implique générer du code et recompiler l'application. L'indépendance face à la machine (puisque c'est le compilateur qui fait la génération de code finale) est définitivement un avantage.

Il y a encore d'autres techniques de convolution rapide qui s'offrent lorsque le problème est suffisamment spécifique et que nous voulons conserver une certaine indépendance face aux noyaux. Par exemple, lorsque nous avons peu de bits par pixel, il est possible de précalculer les « tables de multiplication » pour réduire ainsi le calcul à une référence dans un tableau. Dans le cas de la puce NET32K, les coefficients des noyaux n'ont que trois valeurs admissibles, soit blanc, gris et noir. Les pixels, eux, n'ont qu'un bit. Le très petit nombre de valeurs que l'on peut rencontrer permet une réalisation très efficace de la convolution. Il suffit de précalculer tous les produits d'une rangée du noyau avec les 256 combinaisons de huit pixels qu'elle est susceptible de rencontrer. Lorsque nous rencontrons ces huit pixels, nous n'avons qu'à regarder dans la table précalculée (selon la rangée du noyau) pour connaître la somme partielle.

Puisque nous réduisons une série de huit multiplications par une simple référence, nous divisons au moins par huit le temps de calcul. La génération de la table prend un temps négligeable (environ 55ms sur un 486DX/66Mhz, en Turbo Pascal non optimisé) et le temps total d'application est de l'ordre de 300ms sur une image de 512 x 512 pixels (sur un Pentium 90Mhz, en C/C++ optimisé). Bien entendu, c'est très lent à comparer aux 6.25ms que demande la puce NET32K [Co94] pour exécuter la même tâche, mais c'est tout de même intéressant pour un ordinateur général. C'est présentement cet algorithme qui se trouve au cœur de l'application KernelFinder sous Windows NT que nous avons écrit pour trouver des noyaux intéressant pour les applications NET32K.

L'algorithme n'est intéressant que parce que nous traitons beaucoup de pixels à la fois. Si nous voulions avoir plus de bits par pixel, la taille de la table deviendrait très rapidement encombrante mais cette situation ne provoque pas de perte de performance tant que la table, l'image et les sommes partielles entrent toutes dans la cache du processeur. Comme la performance espérée est directement proportionnelle au nombre de pixels que nous pouvons traiter simultanément, il n'est pas intéressant d'en traiter moins, et en traiter beaucoup plus provoque la saturation puis le débordement de la cache. Si cette solution est

particulièrement intéressante pour les convolutions à la NET32K, elle s'applique peu dans un cadre général.

Bien que ces méthodes puissent être très utiles lorsque les noyaux sont symétriques ou possèdent relativement peu de valeurs différentes dans leurs coefficients, elles deviennent difficilement applicables lorsque les noyaux sont générés par apprentissage. Les noyaux appris ne présentent malheureusement pas de redondance dans leurs coefficients puisque l'apprentissage, en général stochastique, a tendance à générer des valeurs qui apparaissent quelconques, même lorsque les coefficients sont contraints à un intervalle ou encore représentés par des nombres de faible précision. Si nous n'avons ni symétrie ni redondance, le code généré sera toutefois encore efficace, mais nous n'obtiendrons pas de convolution rapide à proprement parler. Ces algorithmes n'offrent donc pas de solution générale.

3.3 Convolutions parallèles.

La prochaine étape que nous envisageons est bien entendu le parallélisme à grande échelle.

Pour produire un algorithme parallèle, il nous faut d'abord déterminer le parallélisme inhérent aux calculs. Supposons que l'on veuille appliquer une convolution sur une image. Pour chacun des points de l'image résultante, correspond une somme de produits. Nous voudrons calculer toutes ces sommes de produits en parallèle, puisqu'elles sont toutes indépendantes, mais, comme dans la plupart des algorithmes parallèles, nous rencontrons un problème de disponibilité des données. Le coefficient du noyau peut être partagé par diffusion mais il faut aussi partager les pixels. Heureusement, ces pixels ne sont pas éparpillés partout dans l'image mais sont tous voisins. Nous allons chercher à amener ces pixels voisins sur le processeur courant. Si on regarde quels pixels doivent visiter un processeur, on voit que la surface qu'il a à parcourir est identique à celle que recouvre un noyau. Nous allons prendre un chemin simple pour parcourir toutes ces cases (voir fig. 5.).

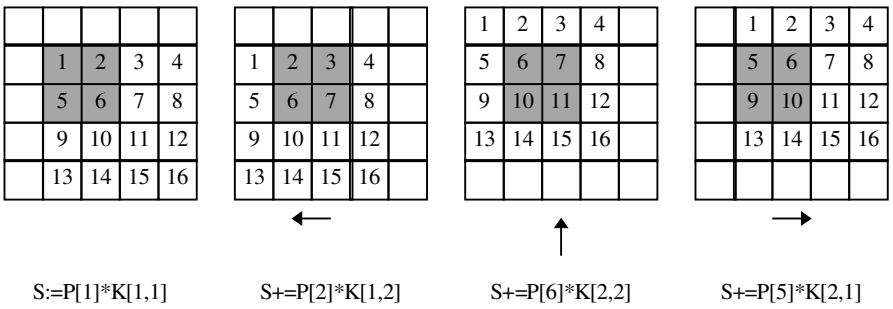


Fig. 5. La migration des pixels dans le calcul d'une convolution avec un noyau 2x2.

Au terme du cheminement, le pixel aura contribué à toutes les sommes partielles qui en avaient usage. À tous les cycles de l'algorithme, nous avons déplacé un pixel, fait un produit et calculé la somme partielle. À la toute fin, nous pourrons appliquer à chaque somme partielle la fonction sigmoïde et stocker le résultat dans les pixels. Les résultats contenus dans la bande de k-1 pixels à droite et en bas de l'image ne seront jamais utilisés.

Présentons maintenant l'algorithme plus formellement. Notons tous les processeurs P_{ij}. Tous les processeurs disposent d'un registre R, qui maintient la valeur du pixel courant, et d'un registre Σ, la somme partielle, préinitialisée à b_i. Tous les processeurs P_{ij} sont connectés à leur quatre voisins, soit $P_{i-1,j}$, $P_{i,j-1}$, $P_{i+1,j}$ et $P_{i,j+1}$. On suppose aussi une connectivité de type tore, comme montré à la fig. 6. Tous les K_{mn}, les coefficients du noyau de largeur k, sont diffusés simultanément via un lien global à tous les processeurs.

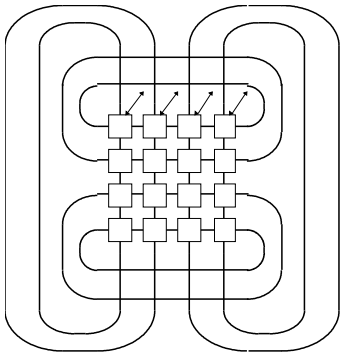


Fig. 6. Un processeur systolique 4x4 et sa connectivité de type torique. Les flèches indiquent des liens avec l'hôte par lesquels entre et sort l'information.

L'algorithme suppose, aussi que les valeurs des pixels sont déjà chargées dans les processeurs P_{ij}. Lorsque toute l'information est en place, on applique l'algorithme de la fig. 7. Le décalage est déterminé ainsi: lorsqu'on décale l'information, les valeurs des registres R sont copiées vers les voisins indiqués par la direction du décalage — cela implique un registre temporaire pour maintenir l'information originale du registre R qui doit être copiée pendant que celui-ci lit l'information provenant du voisin. Le décalage est fait de façon synchrone et sans destruction d'information. Au terme de l'algorithme, tous les registres R des processeurs contiennent le résultat de la convolution, ce que l'on voulait. Maintenant nous disposons d'un algorithme dont le parallélisme est dans l'ordre du pixel (ou du neurone). Or, plutôt que d'augmenter la performance par un petit facteur constant, comme avec les méthodes présentées à la section précédente, nous obtiendrions une augmentation d'un facteur 1024 si nous pouvions avoir un processeur systolique de 32 x 32 éléments de calcul — ce qui correspond à la première étape d'un réseau LeNet 1a. Pour un noyau de 5 x 5, en 25 étapes nous aurions réalisé environ 60000 opérations (environ 20000 décalages, multiplications et additions).

```
convolution()
{
 int direction=gauche;
 busGlobal:=b;
 with all P do Σ:=busGlobal;
 for (i=0; i<k; i++)
  {
   for (j=0; j<k; j++) do
    {
     busGlobal:=K[i][j];
     with all P do Σ:=Σ+R*BusGlobal;
     if (j<k-1) then with all P do decale(direction);
    }
    if (direction == gauche)
             direction=droite;
        else direction=gauche;
    with all P do decale(haut);
  }
 with all P do R:=Sigmoïde(Σ);
}
```

Fig. 7. Algorithme de convolution parallèle en pseudo-C.

3.4 Convolutions parallèles en matériel.

Nous avons introduit un système arithmétique à précision réduite, nous avons discuté de l'utilisation de circuits à logique programmable pour réaliser des processeurs dédiés simples et, enfin, nous avons introduit un algorithme de convolution parallèle. Dans cette section, nous discuterons comment amalgamer ces différents concepts afin de produire un processeur systolique capable de réaliser des convolutions de façon efficace. Nous n'entrerons pas dans le détail (cela fait l'objet du chapitre qui suit) mais nous énumérerons quelques problèmes reliés à la réalisation d'un tel processeur: la connectivité des processeurs, les processeurs complets et processeurs simplifiés et comment généraliser l'algorithme pour une matrice avec un faible nombre de processeurs.

3.4.1 Qu'est-ce qu'un processeur systolique?

Les processeurs systoliques forment une classe particulière de machines SIMD, soit *single instruction, multiple data*, où une seule instruction est appliquée simultanément à plusieurs données. Ce qui rend particuliers les processeurs systoliques, c'est que le flot des données au niveau des éléments de calcul correspond au flot des données dans l'algorithme. Un processeur systolique peut alors être défini

comme un pipeline (potentiellement à plusieurs dimensions) où chaque étage est identique. Chaque étage du pipeline étant identique, le temps passé à calculer dans chacun des éléments est le même. Les échanges d'information dans un processeur systolique sont synchrones. C'est justement à cause de ces déplacements réguliers des données qui peuvent rappeler les battements réguliers du cœur que nous avons des processeurs « systoliques ».

Autre trait important des processeurs systoliques est que chaque unité de calcul n'est généralement conçue que pour une tâche très spécifique. Cela implique donc qu'un processeur systolique en particulier ne s'applique qu'à une classe très restreinte d'algorithmes et d'opérations. Cette surspécialisation permet d'atteindre de très grandes performances au sacrifice de la polyvalence.

Dans notre algorithme de convolution parallèle nous avons défini la connectivité entre nos processeurs P_{ij} de façon à ce qu'il soit possible d'échanger le plus efficacement possible l'information entre ceux-ci. C'est aussi ce que nous avons pour notre processeur systolique, le V.I.P. Chaque unité de calcul communiquera avec ses quatre voisins immédiats (voir fig. 6.) la valeur du pixel qu'il maintient dans son registre. Finalement, chaque élément de calcul ne comportera que les éléments nécessaires au calcul de la convolution.

3.4.2 Connectivité des processeurs et circuits à logique programmable.

Dans un processeur systolique, tous les processeurs ne communiquent qu'avec leurs voisins immédiats. Ils partagent également un bus commun qui leur permet de recevoir les poids et les biais diffusés et des signaux de contrôle qui régissent les processeurs et qui proviennent d'un contrôleur central qui ne fait pas partie de la matrice. Le partage de ces signaux et du bus ne pose pas de problèmes puisque les processeurs n'y ont accès qu'en lecture seulement. Si on se souvient de la fig. 2. qui décrit la connectivité à l'intérieur d'un circuit à logique programmable, on voit que l'architecture est bien adaptée à ce genre de connexions. Chaque bloc peut utiliser en entrée un fil partagé. Cependant, nous avons aussi vu que cette même architecture est plutôt mal adaptée pour un grand nombre de connexions locales. Malheureusement, ce que nous avons le plus dans un processeur systolique, c'est justement des connexions locales! Ce qui nous limitera le plus dans le nombre d'unités de traitement de notre processeur systolique sur circuit à logique programmable ne sera pas le nombre de portes logiques (environ 20000 pour les plus gros circuits à logique programmable Altera) mais les connexions qu'il sera possible de faire entre les processeurs.

3.4.3 Processeurs simplifiés.

L'attrait principal d'un processeur systolique est la régularité de la conception. Tous les processeurs sont habituellement identiques. Donc, dès qu'on parvient à réaliser un élément de calcul fonctionnel, tous les autres le sont aussi. Si nous utilisons des éléments de calculs identiques, nous utiliserons aussi des éléments de calcul complets dans la bande de k-1 pixels pour lesquels aucun résultat valide n'est calculé. Les résultats que calculent ces processeurs ne seraient valides que si l'on considère l'image comme un tore, ce que nous ne faisons pas. Puisque ces résultats ne nous intéressent pas il est clair que réduire ces processeurs à la seule la fonction d'emmagasinage de l'information dans le registre R est suffisante pour permettre aux autres processeurs de bien calculer leurs résultats. Les processeurs qui ne servent qu'à transmettre de l'information sont ainsi réduits à deux registres (le registre R et le registre temporaire). Nous nous épargnons aussi les connexions vers le bus de diffusion et ne conservons que quelques bits de contrôle.

3.4.4 Sur les limites des circuits à logique programmable.

Nous avons déjà vu qu'une limitation importante dans les circuits à logique programmable Altera était le type de connexions entre les cellules logiques. Les circuits Altera ne permettent que des connexions globales que nous épuisont rapidement avec certains types de circuits. Cette limitation n'est cependant pas universelle: d'autres fabriquants offrent des puces avec des caractéristiques différentes. Nous avons déjà dit que Xilinx, par exemple, permettait de segmenter les connexions mais qu'en revanche, cela devenait très compliqué de prévoir les délai avec précision. Puisque nous avons des puces Altera montées sur la carte V.I.P., nous avons rencontré des problèmes de routage même lorsque les circuits étaient plutôt simples.

Autre limitation importante est le nombre de circuits à logique programmable que l'on peut s'offrir. Sur la carte V.I.P, dont nous parlerons plus longuement au chapitre 4, nous avons 6 circuits à logique programmable Altera. Un sert de contrôleur d'entrées/sorties, un de contrôleur systolique alors que les quatre autres gèrent la matrice de calcul. Cela met environ 80000 portes logiques universelles à notre disposition pour la matrice d'unités de traitement et la connectivité sur la carte elle-même permet dans une certaine mesure de faire abstraction du fait que les circuits à logique programmable sont des entités séparées. On peut concevoir le groupe de quatre circuits à logique programmable comme un « super circuit à logique programmable ». Cela correspond à la situation démontrée à la fig. 8.

Évidemment, nous prenons soin de ne pas séparer une même unité de calcul entre deux circuits car les communication interpuces sont quand même moins efficaces que les communications intrapuces. Il faut aussi considérer que les puces n'offrent quand même pas un très grand nombre de pattes disponibles pour le concepteur. Beaucoup d'entre elles sont utilisées pour la programmation du circuit, pour l'alimentation et autres signaux dédiés. Pour une puce de 300 pattes, il ne faut pas compter sur plus de 200 pattes. De plus, il faudra aussi souvent multiplexer des signaux pour économiser le matériel. Dans certains cas, le multiplexage ainsi qu'une mise en pipeline ne nuisent nullement à la performance, alors que dans certains autres la mise en pipeline est plutôt difficile à réaliser et nous observons une perte de performance.

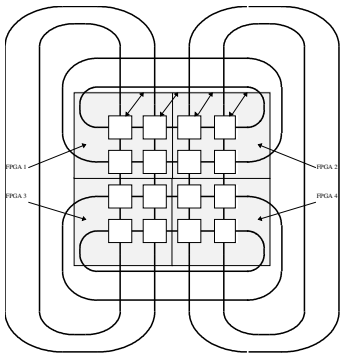


Fig. 8. Un processeur systolique 4x4 sur 4 circuits à logique programmable.

Chapitre 4: Réalisation d'algorithmes parallèles sur ordinateurs à logique programmable

Dans ce chapitre, nous présenterons l'environnement V.I.P. et ses différentes composantes matérielles ainsi que les outils logiciels que nous avons developpé. Nous présenterons tour à tour les différentes composantes de l'environnement — compilateur systolique, simulateur, debugger et autres outils d'aide à la conception. Bien que nous nous attardions dans ce chapitre à décrire ce que nous avons réalisé, nous voulons attirer l'attention du lecteur sur le fait que l'application présentée dans ce mémoire, à savoir les convolutions en deux dimensions, n'est en fait qu'une des nombreuses applications qui seraient réalisables sur le processeur V.I.P. Le processeur V.I.P. permet des réalisations variées même si certains détails de son architecture le prédisposent tout naturellement au calcul des convolutions en deux dimensions. Les outils que nous avons sont axés sur le traitement d'image, mais il est tout aussi possible de réaliser d'autres environnements pour d'autres applications.

4.1 Environnement de développement.

Lorsque nous optons pour un modèle particulier, nous sommes habituellement convaincus de sa viabilité. Pour un microprocesseur, la viabilité ne se définit pas seulement en terme de la puissance qu'il peut offrir mais on considère aussi les différents environnements capables d'être supportés par celui-ci. Un fabriquant de microprocesseurs, par exemple Intel, n'offrira jamais un nouveau produit isolé. Si c'est un microprocesseur, on pourra trouver chez Intel tout ce dont on aura besoin (contrôleur de cache,

d'interruptions, système d'exploitation minimal, etc.) afin de rapidement pouvoir l'intégrer à une application. Si le fabriquant n'offre pas ces outils, tout développement devient plus difficile et rend le produit nettement moins attrayant.

C'est aussi vrai pour une carte d'extension comme la carte V.I.P. Il faut pouvoir offrir à l'utilisateur les différents outils qu'il espère trouver dans un environnement de développement. Cela permet d'exploiter plus rapidement et plus efficacement la fonctionnalité du processeur à partir d'un modèle de haut niveau.

4.2 L'architecture et l'utilisation de la carte V.I.P.

Avant de poursuivre, il serait utile de bien décrire le processeur V.I.P. afin de mieux situer chacun des outils que nous présenterons par la suite. La réalisation matérielle dépend évidemment de l'application qui nous intéresse. Dans notre cas, nous ne voulions pas un système indépendant mais un coprocesseur. Le choix de l'hôte dépend principalement des outils logiciels que nous y trouvons déjà et de la disponibilité du système lui-même. L'interface, si possible, doit convenir à un plus grand nombre d'hôtes possible, afin qu'il soit possible éventuellement de transporter le matériel vers une autre plate-forme sans le modifier. Le standard d'interface au bus PCI, par exemple, est utilisé par de nombreux constructeurs (les constructeurs PC, Apple, DEC, etc.). Le choix de l'hôte sera déterminé par le logiciel disponible sur cette plate-forme, son coût et d'autres considérations pratiques. Nous avons opté pour le PC pour des raisons de disponibilité et de coût.

4.2.1 Architecture de la carte V.I.P.

Le processeur V.I.P. contient deux unités principales. En premier lieu, nous avons un contrôleur d'entrées/sorties qui gère les transactions entre la carte et l'hôte via le bus PCI. Ce contrôleur est composé d'une partie d'une puce dédiée fabriquée par AMCC qui gère les signaux du bus. L'autre partie du contrôleur est un circuit à logique programmable à mémoire non volatile. Cette puce permet entre autres de programmer les autres circuits à logique programmable et de simplifier la communication entre l'hôte et les autres circuits. Bien qu'éventuellement reprogrammable, sa non-volatilité peut poser un problème lorsqu'on veut reprogrammer tout à la volée. Il est donc préférable que sa programmation demeure la plus générale possible afin d'accommoder le plus grand nombre d'applications sans nécessiter la reprogrammation qui exige l'extraction de la puce et l'utilisation d'un programmeur d'EPLD.

En second lieu, nous avons cinq circuits à logique programmable qu'il nous est possible de programmer à la volée. Ces cinq circuits ont accès à leur propre banque de mémoire. Le premier, cependant, possède deux banques. Cela a été originalement pensé de façon à permettre une architecture Harvard pour ce circuit qui devait être dédié au contrôle du processeur systolique. Ses deux canaux de 32 bits peuvent être combinés pour avoir des instructions ou des données de 64 bits. Pour plus de détail sur la carte V.I.P, consultez [Cl96].

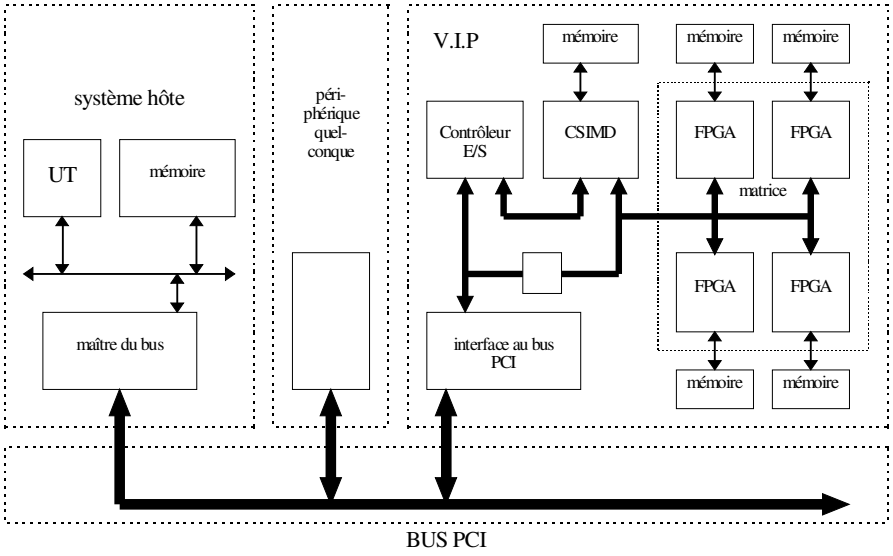


Fig. 9. L'hôte, le système V.I.P. et ses principales composantes.

4.2.2 Utilisation de la carte V.I.P.

Lors de la mise sous tension, seul le contrôleur d'entrées/sorties est fonctionnel. Ce dernier permet de programmer les autres circuits à partir d'un flot de données qui provient de l'hôte. Les données nécessaires à la programmation sont générées par le compilateur de matériel, dans notre cas le compilateur Altera.

Cette description matérielle est produite à partir d'un programme écrit dans le langage de haut niveau ADHL. Cette phase est assez ardue. Il nous faut traduire notre algorithme en circuits logiques et il nous faut tenir compte de nombreuses contraintes telles que l'assignation des pattes des puces, du protocole de communication avec le contrôleur d'entrées/sorties, etc.

Dès que nous avons cette description, il suffit alors de l'écrire dans un espace de mémoire réservé. À cet espace de mémoire réservé, correspond une adresse que le contrôleur reconnaît comme l'espace de configuration. Le contrôleur d'entrées/sorties convertit cet accès mémoire en séquences de contrôles pour la programmation des circuits et transfert les bits provenant du bus aux circuits à logique programmable. Tout cela se fait sans le concours de l'utilisateur. La programmation est déclenchée automatiquement à chaque fois qu'un accès en écriture est fait dans cet espace mémoire. Une fois que la description du matériel est complètement transférée dans la carte, les circuits sont fonctionnels et prêts à exécuter le programme.

Le programme, de la même façon que la description matérielle, est simplement copié dans un autre espace mémoire réservé de la carte. Le contrôleur d'entrées/sorties, encore, fait la conversion d'adresses afin que le programme soit copié dans la mémoire de programme du contrôleur systolique. Le programme est produit à partir d'un langage de très bas niveau (qui est très similaire au microcode) qui est assemblé et compilé par le programme SAS (dont nous reparlerons). Les données sont aussi copiées directement en mémoire, de la même façon. Chaque banque de mémoire est accessible en lecture et en écriture par l'hôte. Pour charger les données, il suffit d'écrire en mémoire et de lire cette même mémoire pour les récupérer. Ensuite, il ne reste qu'à donner des ordres de haut au niveau contrôleur systolique, comme démarrer en exécutant tel sous-programme, arrêter, etc.

4.2.3 Architecture du contrôleur systolique

Le contrôleur systolique à pour tâche d'orchestrer toutes les opérations qui sont réalisées par la matrice de processeurs. Si on réduit l'algorithme de convolution parallèle présenté au chapitre précédent en instructions atomiques, on remarque que certaines instructions de contrôle sont indépendantes. L'activité des registres qui maintiennent les sommes partielles est indépendante des décalages, pour ne donner que cet exemple. Si on examine minutieusement les instructions nécessaires pour réaliser l'algorithme, nous allons constater qu'en tout, nous pouvons exécuter jusqu'à dix-sept instructions différentes simultanément. En fait, ce que nous avons, c'est près de cent instructions différentes qui se divisent en dix-sept groupes indépendants. Les instructions d'un même groupe affectent les mêmes unités fonctionnelles et ne peuvent donc pas être exécutées simultanément.

Il existe plusieurs façons de réaliser matériellement ce contrôleur. Une approche pourrait demander un processeur presque général, avec une banque de registre, un séquenceur et une unité arithmétique et logique. Cependant, cette approche, quoique classique, présente l'inconvénient majeur de couper presque tout le parallélisme que nous avons trouvé au niveau de l'algorithme. Dans ce modèle, les instructions sont exécutés séquenciellement (ou à un très faible degré de parallélisme, s'il existe un ou plusieurs pipelines) et nous perdons beaucoup de puissance. L'autre solution, c'est l'approche very long instruction word, ou VLIW, où toutes les unités fonctionnelles du processeur opèrent en parallèle et où les instructions contrôlent toutes les unités simultanément. Cela permet de réaliser pleinement le parallélisme inhérent au contrôle de l'algorithme. Ainsi nous aurons dix-sept de ces unités.

De plus, au niveau matériel, comme nous n'avons que des opérations et des registres spécialisés, cela nous permet d'optimiser le circuit et d'éliminer les mécanismes qu'une approche plus générale aurait demandés. Nous n'avons pas d'unité arithmétique et logique générale mais plutôt une banque de registres à fonctions. Certaines instructions ne consistent qu'en bits de contrôle que l'on passe directement à la matrice systolique. Les autres instructions affectant les registres internes ne demandent pour la plupart qu'un incrémenteur, ce qui peut être façilement jouxté à un registre. Pour une liste et une description de chaque instruction, voir l'annexe 2.

Les registres internes sont nécessaires pour conserver l'état du contrôleur systolique qui exécute le programme. Lorsqu'on considère la fig. 10, on voit que les unités fonctionnelles sont indépendantes et communiquent entre elles par un bus local (par opposition au bus global qui relie le contrôleur systolique au contrôleur d'entrées/sorties et à la matrice) et c'est l'utilisation de ce bus local qui détermine quelles opérations s'excluent mutuellement. Puisque les instructions ne sont pas encodées, nous n'avons plus

besoin de microcode et les décodeurs sont réduits à un minimum dans le processeur. Les décodeurs servent principalement à gérer les deux modes d'accès aux registres (par l'hôte ou par le programme).

Des instructions qui ne sont pas encodées (ou très peu) correspondent à la méthode VLIW, où chaque groupe de bits de l'instruction contrôle directement une unité fonctionnelle du processeur. Comme toutes les instructions sont suffisamment simples (ou dont chaque partie est exécutée en parallèle) pour être exécutées en un seul cycle, nous n'avons pas besoin de séquenceur.

4.2.4 Assembleur systolique.

Vu le nombre très limité d'instructions, il est facile d'écrire un compilateur pour ce processeur. Toutefois, un langage de haut niveau est impensable puisque nous n'avons pas un processeur général susceptible d'exécuter une vaste gamme de programmes, mais un processeur très spécialisé qui ne peut qu'exécuter que les applications peu nombreuses réalisables sur notre processeur systolique. Un assembleur semble être un choix raisonnable à ce stade, considérant que le contrôleur est bien défini.

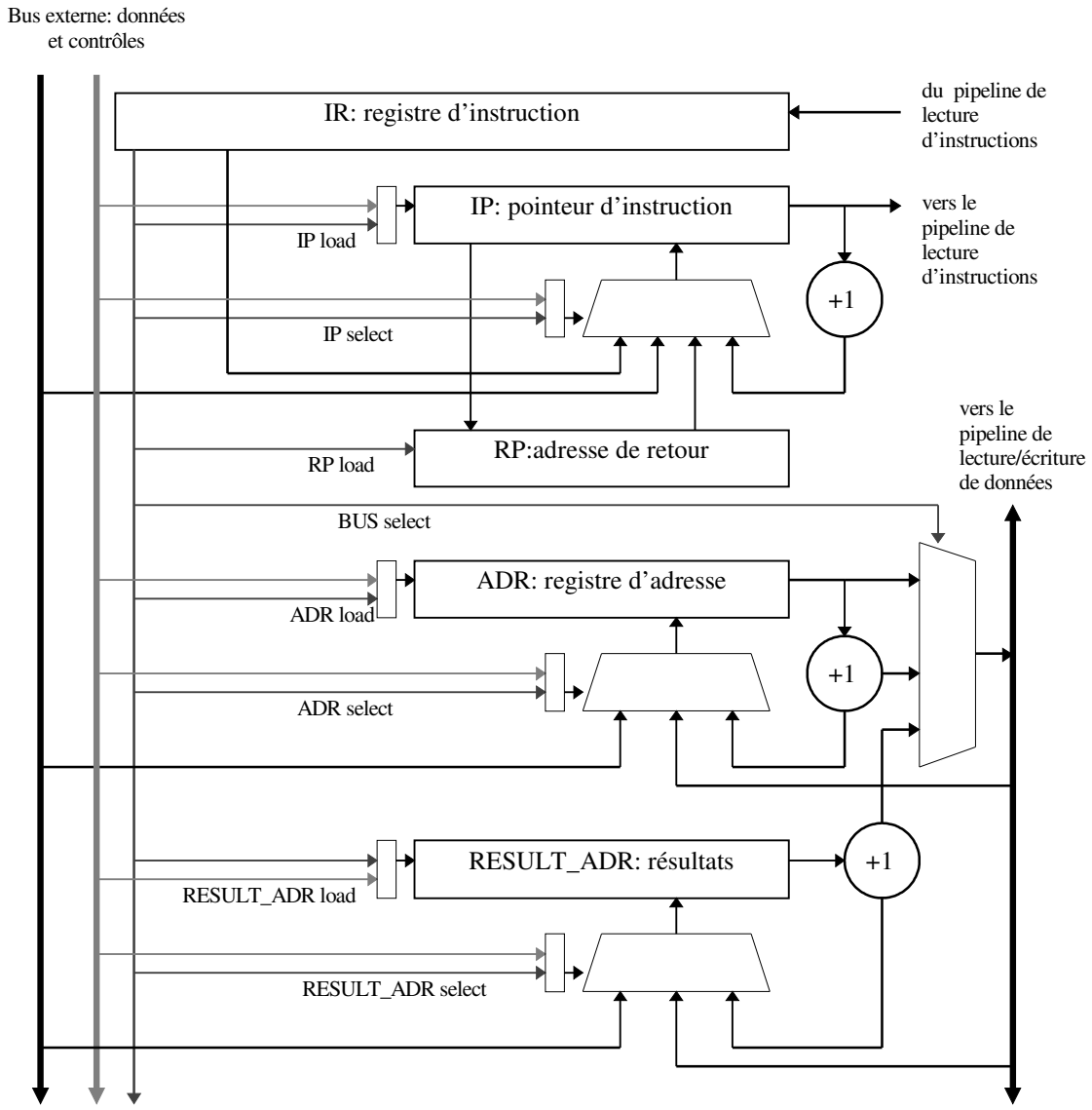


Fig. 10. Quelques unités fonctionnelles indépendantes dans le contrôleur systolique.

Lorsqu'on examine les instructions nécessaires pour l'exécution de l'algorithme, ce que l'on constate, c'est que l'on a des séries d'instructions qui sont totalement dédiées à une partie du processeur. On pourrait regrouper les instructions qui sont les plus souvent pairées en « superinstructions » représentées par des mnémoniques simples, mais cela limiterait l'accessibilité au matériel. Donc, nous avons opté pour un assembleur qui ressemble aux assembleurs classiques: nous avons beaucoup d'instructions différentes, on peut avoir des fichiers inclus, des étiquettes, des appels de procédure et des commentaires. Mais, contrairement aux assembleurs communs où chaque mnémonique suivie de ses paramètres est complète en elle-même, nous pouvons combiner les instructions en une superinstruction en

énumérant simplement les instructions que l'on veut exécuter en les séparant par une virgule. Les superinstructions sont séparées par l'inévitable « **;** ».

L'assembleur génère un fichier binaire que l'on peut télécharger dans la mémoire de programme du contrôleur systolique. Le code n'est pas relatif — parce que le contrôleur systolique ne supporte pas ce mode d'adressage — mais on peut combiner plusieurs programmes en les concaténant et comme l'assembleur génère un *map file* détaillé, l'utilisateur (et le debugger) peut connaître les adresses des points d'entrée pour les utiliser comme valeur initiale du pointeur d'instructions (IP).

L'assembleur SAS (pour Sytolic Assembler) est décrit en détail dans l'annexe 2, où l'on présente les différents éléments du langage présentement défini. Une caractéristique importante de SAS est sa *reconfigurabilité*. Le langage qui doit être reconnu et le code qui doit en être généré peuvent lui être décrits dans un fichier séparé. Ainsi il est facile de replacer le jeu d'instruction s'il advenait que nous décidions d'une autre architecture pour le contrôleur systolique. SAS présente toutefois des limitations. S'il peut gérer correctement plusieurs unités fonctionnelles, il suppose présentement qu'elles sont toutes synchrones. S'il advenait qu'une instruction soit plus longue à exécuter que les autres, il faudrait inclure un module supplémentaire pour gérer le séquencement. De plus, SAS suppose quand même une certaine uniformité syntaxique mais comme il est basé sur une coquille d'analyseur syntaxique et de générateur de code, il est relativement aisé d'implanter de nouvelles structures syntaxiques, d'ajouter le séquencement ou d'y faire d'autres améliorations.

4.3 Simulateurs systoliques et debugger.

Une solution intéressante pour le développement de logiciel sur un processeur dédié serait d'avoir un simulateur logiciel suffisamment fidèle à la réalité matérielle de ce processeur pour que l'on puisse utiliser la carte ou le simulateur indifféremment. Le simulateur pourrait permettre à plusieurs personnes de développer du logiciel alors que peu d'exemplaires de la carte sont disponibles.

Nous devons distinguer deux classes de simulateurs. Le premier type simule le matériel seulement dans le but de le remplacer (niveau comportemental), alors que le second type offre aussi un environnement complet, une machine virtuelle (niveau structurel).

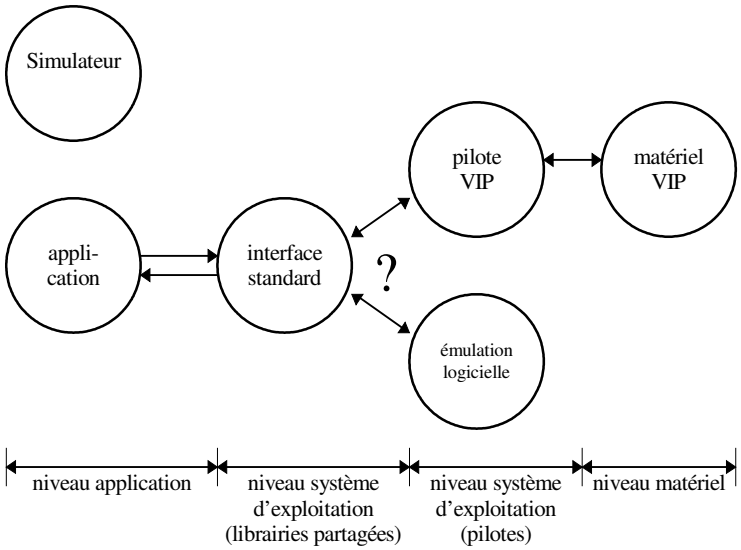


Fig. 11. Un simulateur, un API, un émulateur et matériel.

Lorsqu'on considère le premier type de simulateur, on parle généralement d'émulateur. Un émulateur se contente de remplacer le matériel absent et sa réalisation logicielle ne reflète pas nécessairement les mécanismes rencontrés dans le matériel. Au contraire, les mécanismes de l'émulateur sont réalisés de façon à avoir la plus grande performance possible. L'émulateur n'offre pas d'environnement complet pour l'usager mais seulement un substitut de matériel accessible à travers une couche d'abstraction du matériel utilisable par les applications au travers d'une application program interface, ou API (voir fig. 11). L'émulateur de ce type que l'on rencontre le plus souvent est l'émulateur de coprocesseur numérique qui réalise en logiciel les calculs en virgule flottante de façon transparente pour les programmes.

Le second type de simulateur ne se contente pas de remplacer le matériel mais réalise aussi tous les mécanismes de celui-ci. Ce type de simulation se veut le plus près de la logique interne du matériel qu'il simule, et offre, au travers d'un environnement virtuel, toute l'information que l'utilisateur peut trouver intéressante. Par exemple, à partir de l'hôte, en matériel, nous avons accès à certains registres alors que d'autres demeurent inaccessibles. Un simulateur de base pourrait n'offrir que la visibilité réduite qu'offre l'interface matérielle, mais, idéalement, il devrait plutôt donner toute l'information disponible au niveau structurel et rendre aussi disponible l'information qui est utilisée dans les couches du matériel qui restent ordinairement inaccessibles pour l'utilisateur, comme certains registres auxquels nous n'avons pas directement accès. Puisqu'on ne s'intéressera qu'au déplacement de l'information, une simulation au niveau du transfert de registre semble suffisante. Descendre plus bas, par exemple au niveau des portes logiques, est dans ce contexte complètement inutile.

Le simulateur, qui exécute le programme de l'utilisateur, doit offrir le meilleur temps d'exécution possible ou, lors des séances de débuggage interactif, offrir un temps de réponse idéal. En fait, l'utilisateur doit pouvoir considérer comme « instantanée » l'exécution d'une instruction en mode interactif et « satisfaisant » le temps d'exécution d'un programme complet, de façon à minimiser le temps total de développement. Le simulateur devra aussi, bien entendu, accepter les mêmes fichiers pour les programmes exécutables, c'est-à-dire ceux générés par SAS, et les données encodées de la même façon, c'est-à-dire provenant de la base NIST (voir l'annexe 1 à ce sujet).

Finalement, bien que l'on puisse trouver discutable la nécessité d'une interface graphique et conviviale, notre propre expérience semble démontrer que, bien au contraire, un environnement graphique dans lequel l'information est bien présentée et où les commandes sont facilement repérables aide l'usager à se familiariser rapidement avec le système. Cet aspect est d'autant plus important que les systèmes d'exploitation courants offrent tous une interface graphique évoluée. Le simulateur devrait donc s'intégrer harmonieusement avec le reste des outils que l'utilisateur utilise déjà, selon son système d'exploitation (voir fig. 12).

L'interface de notre simulateur offre tous les outils d'un debugger. Il permet d'une part d'exécuter un programme, bien entendu, mais aussi d'observer le programme graphiquement. La matrice de processeurs est présentée de façon à ce que l'utilisateur puisse observer la migration des pixels et l'état des registres Σ des processeurs de la matrice. Il lui est aussi possible d'observer un processeur en particulier en utilisant une représentation numérique plutôt que graphique, tout comme d'ausculter les registres et de les modifier au moment de l'exécution. Il peut aussi exécuter le programme en pas à pas ou en continu, selon son choix. Le programme qui s'exécute instruction par instruction est présenté sous sa forme binaire et sa forme mnémonique, si l'utilisateur le choisit. Le simulateur (comme le matériel) supporte les points de contrôle (*breakpoints*) qui permettent d'exécuter rapidement le programme jusqu'à ce que l'on atteigne un point intéressant du programme. Sur un point de contrôle, le simulateur tombe automatiquement en mode pas à pas. L'utilisateur peut relancer le programme s'il le désire, et même désactiver les points de contrôle. Finalement, nous n'avons pas voulu fournir un outil minimal pour réaliser des programmes V.I.P., mais un environnement le plus complet possible en considérant le temps dont nous disposions.

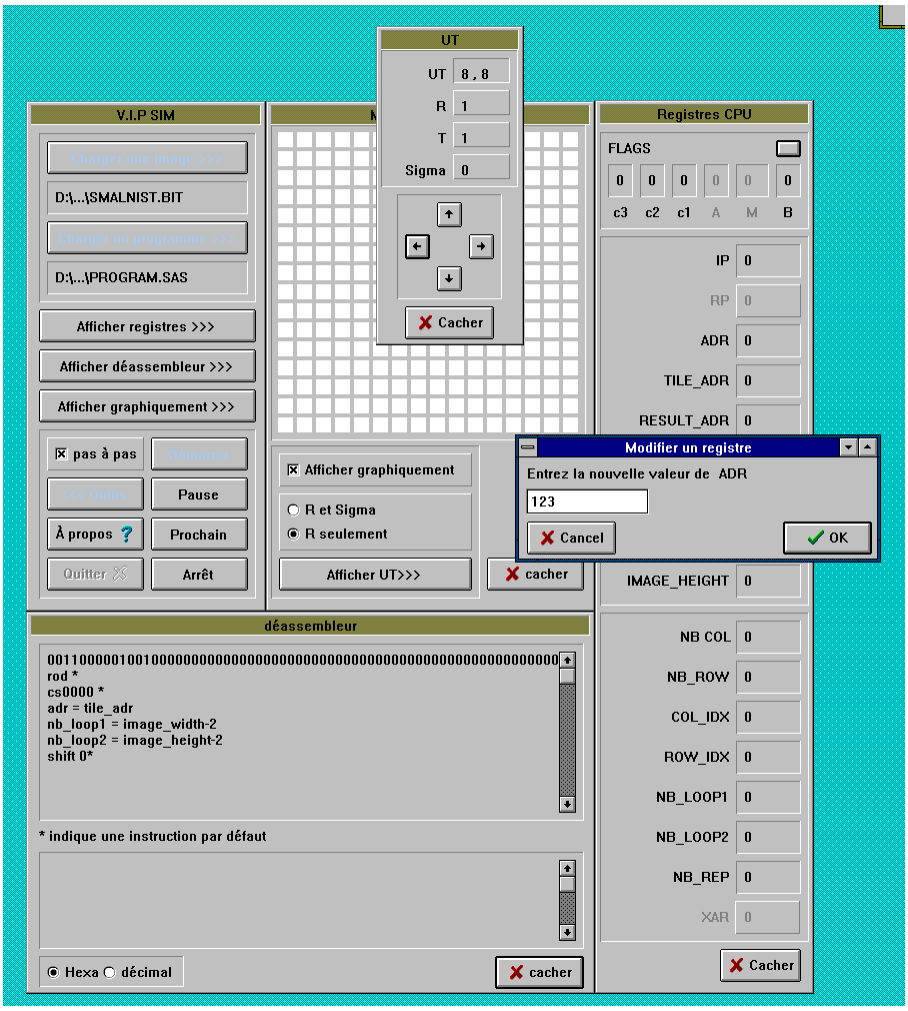


Fig. 12. L'environnement graphique du simulateur V.I.P.

4.4 La matrice systolique.

Nous devons prendre soin de bien faire la distinction entre les deux sortes d'unités de traitement que nous sommes susceptibles de retrouver dans un processeur systolique pour un réseau de neurones artificiels. Nous pouvons n'avoir que des unités de traitement qui ne calculent que la convolution et la fonction sigmoïde. Ces unités seront plus simples que celles dont on a besoin pour faire l'apprentissage à l'aide du processeur. Nous n'avons pas présenté l'algorithme parallèle d'apprentissage, mais nous pouvons dire qu'il est nettement plus complexe, en terme de migration de données, que l'algorithme de convolution. Nous présenterons dans cette section les deux types d'unités de traitement. Nous décrirons en détail les unités pour l'évaluation et nous énumérons les caractéristiques des unités pour l'apprentissage.

4.4.1 Une unité de traitement pour l'évaluation.

Considérons maintenant la réalisation matérielle d'une unité de traitement (UT) capable de faire les calculs pour l'évaluation d'un réseau de neurones. Tout d'abord, l'UT doit être capable de transporter l'information vers ses voisins. Nous avons vu que les glissements ne se font seulement qu'en trois directions (se rappeler la fig. 5.). Lors du chargement initial, nous avons besoin, pour une rangée d'unités, d'une quatrième direction qui permet de lire du bus ou de la mémoire les valeurs que nous faisons glisser dans la matrice. On se rappellera que lorsque nous avions parlé du prétraitement, nous avions supposé qu'il avait déjà été fait au moment où on commence la convolution. Le prétraitement consiste en fait à charger une colonne d'information, faire glisser cette information d'une colonne vers la droite, puis recommencer jusqu'à ce que toute la matrice soit chargée. L'information provient initialement de la mémoire de la matrice, où elle a été écrite par le programme d'application. Nous allons récupérer l'information après la convolution en sens inverse: lire une colonne, décaler vers la gauche et recommencer jusqu'à ce que toute l'information soit récupérée. L'information récupérée est réécrite dans la mémoire de la matrice que le programme d'application, qui s'exécute sur l'hôte, peut lire.

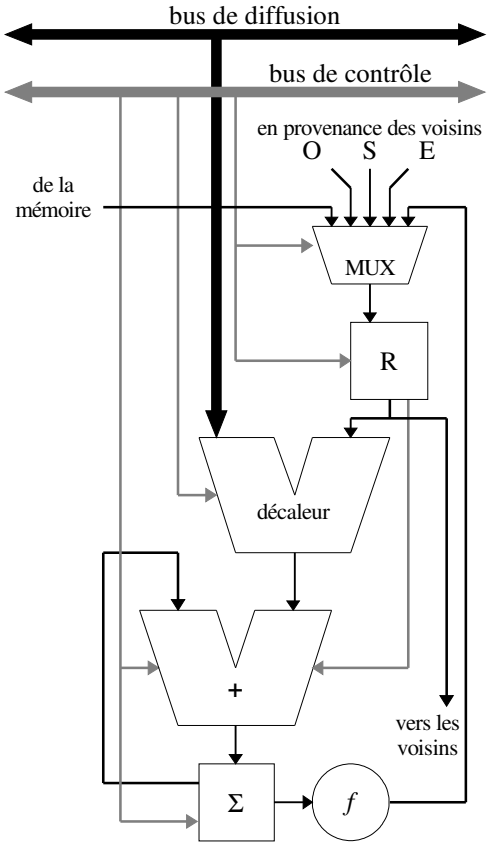


Fig. 13. Les principaux éléments d'un UT pour l'évaluation. Les chemins en noir désignent les données, alors que les chemins en gris indiquent des signaux de contrôle.

Pour ce qui suit, considérez la fig. 13. Le décaleur, comme nous l'avons expliqué précédemment, ne réalise que des décalages vers la droite, ce qui le simplifie grandement. Nous ne devons cependant pas oublier qu'il doit aussi maintenir correctement le signe du résultat: les poids et les activations peuvent être positifs ou négatifs. La sortie du décaleur est directement additionnée à l'accumulateur Σ, qui maintient la somme partielle. La fonction *f* de la fig. 13 est la sigmoïde (telle que décrite à la section 2.1), $\tilde{\sigma}_j(\tilde{s}_j)$, et où $\tilde{s}_j = \Sigma$. Ensuite, le séquencement des opérations internes se fait à partir du programme où chaque instruction contient les bits pour contrôler les UT.

La complexité de la fonction $\tilde{\sigma}_j(\tilde{s}_j)$ peut poser un problème. Si nous utilisons 3 bits pour les activations, nous sommes libres de choisir huit points pour échantillonner la fonction sigmoïde. Mais la forme de la fonction et les points d'échantillonnage peuvent difficilement être arbitraires; il nous faut avoir une fonction qui est rapide à calculer et qui occupe le moins d'espace possible. Si l'échantillonnage est régulier nous utilisons moins de bits (et moins d'opérations) pour déterminer les intervalles que si l'échantillonnage est quelconque et dense en quelques régions. La fonction utilisée est la fonction présentée au chapitre 2.

Voyons maintenant comment il est possible de calculer l'activation d'un neurone à partir du circuit logique présenté à la fig. 13. Rappelons-nous l'algorithme de la section 3.3. Supposons, encore, que le prétraitement ait déjà été effectué. Pour toutes les instructions, les bits de contrôle proviennent du contrôleur systolique. Nous allons d'abord diffuser le biais. Comme nous n'avons qu'un seul biais par convolution, c'est une opération que nous pouvons ne faire qu'une seule fois. Le biais arrive dans chaque UT par le bus global, passe dans le décaleur sans être affecté, puis est additionné au registre Σ, qui a été remis à zéro préalablement. Chaque registre R contient déjà un pixel. Ensuite, on diffuse un poids. Le poids est décalé par la valeur de R, et est additionné au registre Σ. On procède au décalage en choisissant l'entrée correspondante à la direction du décalage par le multiplexeur et on charge le registre R avec cette nouvelle valeur. On répète jusqu'à ce que tous les poids de ce noyau aient été diffusés. Lorsque c'est fait, on choisit l'entrée qui correspond à la fonction *f* et on charge cette valeur dans R. Il ne restera plus par la suite qu'à récupérer l'information.

Il faut dire que pour le prétraitement et la récupération des données c'est le contrôleur systolique qui gère la lecture et l'écriture en mémoire. L'adresse est générée par celui-ci et seulement les données parviennent aux UT. Le nombre de bits par pixel influe directement sur le nombre de UT qui peuvent accéder à la mémoire simultanément. Puisque chaque canal n'a que 32 bits (à raison d'un canal par circuit)

et que nos pixels ont 3 bits, nous ne pouvons charger (ou sauvegarder) plus de 10 pixels par cycle. Ce n'est pas vraiment limitatif puis que nous n'avons, comme nous l'avons vu, que 9 UT par circuit.

4.4.2 Un UT pour l'apprentissage.

Les UT pour l'apprentissage sont plus complexes que les UT pour la passe avant seulement. Nous nous limiterons aux rétroconvolutions (c'est-à-dire la rétropropagation de l'erreur dans les convolutions), mais le modèle reste globalement valide en ce qui concerne la rétropropagation de l'erreur en général. Souvenons-nous des équations déjà présentées. Nous devons calculer:

$$g_i = \tilde{\sigma}'_i(\tilde{s}_i) \sum_{j \in succ(i)} w_{ij} \tilde{g}_j \quad \text{ainsi que} \qquad (1)$$

$$\Delta w_{ij} = -\tilde{\eta} \tilde{x}_i g_j \qquad (2)$$

où: g_i est un gradient, sur 16 bits.
$\tilde{g}_j$ est un gradient, sur 3 bits, calculé à partir de $\delta(g_j)$
w_{ij} est un poids sur 16 bits.
$\tilde{\sigma}'_i(x)$, la dérivée de la sigmoïde, est sur un bit.
Δw_{ij} est sur 16 bits.
$\tilde{\eta}$ est le pas de gradient, en puissance négative de deux, sur 6 ou 8bits.

Les équations (1) et (2) doivent être calculées efficacement pour rendre l'apprentissage intéressant en matériel. L'algorithme d'apprentissage est beaucoup plus complexe que la simple convolution comme on le constate lorsqu'on considère la migration nécessaire des données dans le processeur systolique.

Nous ne présenterons pas l'algorithme en détail, mais nous le décrirons dans ses grandes lignes. L'algorithme de rétroconvolution ne diffère pas beaucoup de l'algorithme de convolution présenté à la section 3.3. Il demande seulement plus de transport de données. Lorsqu'on considère l'équation (1), on voit que nous avons besoin des poids et des gradients. On peut calculer efficacement les nouveaux gradients pour chacun des pixels en utilisant le registre Σ car les gradients discrétisés (en puissances de deux) $\tilde{g}_j$ sont du même format que les pixels (donc transportables de la même façon) et les poids peuvent être diffusés. Il nous faut aussi une fonction pour calculer la dérivée de la fonction sigmoïde (représentée par *f'* dans la fig.14). Par la suite, il faut récupérer ces gradients non discrétisés, les g_j, ainsi que les

activations pour calculer l'équation (2). Finalement, il faut déplacer les Δw_{ij} pour procéder à la mise à jour des poids. Un UT qui serait capable de calculer les équations (1) et (2) ressemblerait à celui présenté dans la fig. 14.

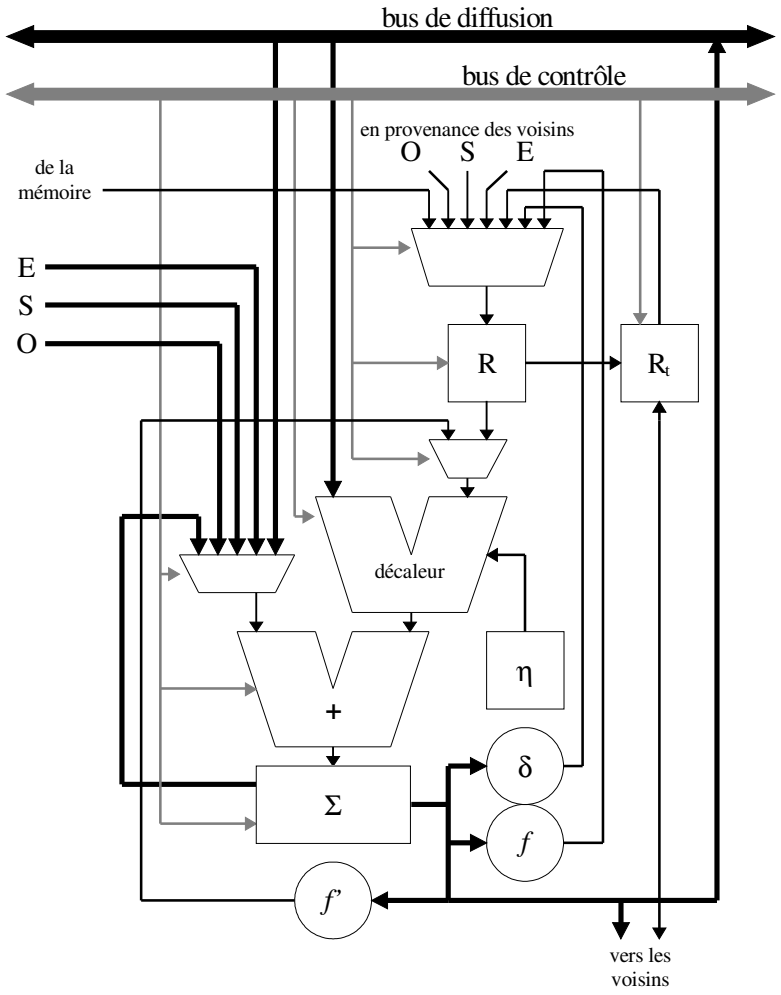


Fig. 14. Une unité traitement pour l'évaluation et l'apprentissage.

Considérons maintenant les éléments de ce nouvel UT. Le circuit combinatoire qui calcule la dérivée est beaucoup moindre, en terme de complexité, que celui qui calcule la sigmoïde. Nous avons le même espace d'entrée mais nous n'avons que deux valeurs possibles en sortie (donc un bit). Si la somme est dans l'intervalle]-1,1[, nous avons une dérivée de 1 et de $^1/_{32}$ autrement. Nous avons aussi ajouté η au multiplicateur. Maintenant, le décaleur doit non seulement utiliser l'activation mais aussi η pour décaler les $\tilde{g}_j$. Le fait que η soit toujours négatif (quoique les signes s'annulent dans les équations) ne réduit pas de beaucoup la complexité du circuit parce que l'influence de η est conditionnelle. Nous n'utilisons pas le

pas de gradient à toutes les étapes, mais seulement celles utilisées pour calculer les changements de poids. Le second registre, R_t , peut être utilisé pour nous épargner la gymnastique d'amener les informations en mémoire et de les y récupérer par la suite, mais il peut aussi être très coûteux de l'utiliser car il occupe beaucoup d'espace (si les activations et les gradients discrétisés n'occupent que 3 bits, les gradients et les poids en occupent 16).

Si on regarde attentivement la fig. 14, on constate qu'un UT capable de l'apprentissage comme de l'évaluation est beaucoup plus complexe que l'UT pour l'évaluation simple. De façon réaliste, nous avons trouvé que deux UT capables d'apprentissage et de l'évaluation peuvent entrer sur un circuit à logique programmable Altera, mais au coût de multiplexer et de pipeliner la plupart des unités. Si nous ne pouvons disposer que d'un tel UT par circuit à logique programmable, le gain en vitesse est très rapidement perdu dans la gestion du transport de l'information, autant au niveau de la communication hôte-processeur systolique qu'au niveau de la gymnastique qui devient nécessaire pour simuler une matrice plus vaste. Nous estimons que si nous ne pouvons pas disposer d'au moins 16 UT, l'apprentissage sera plus rapide en logiciel. D'abord, parce qu'il faut passer par le pilote de la carte, qui transfère des données via le bus PCI relativement lent (33Mhz alors que le processeur est certainement cadencé à plus de 100Mhz), pour enfin arriver à la carte V.I.P qui ne fonctionne qu'à une cadence réduite de 16Mhz. Sur un Pentium, une multiplication pipelinée ne prend qu'un cycle. Si on ne compte pas le coût relié au transport de l'information, on se retrouve avec des UT qui sont environ 6 fois plus lents… Il faudrait que ça soit au moins deux fois plus vite pour que ça soit intéressant, donc 16 serait à peu près le nombre de UT nécessaires pour commencer à avoir des performances intéressantes par rapport à un Pentium 100Mhz.

Chapitre 5: Résultats.

Dans ce chapitre, nous présenterons les résultats que nous avons obtenus au cours de nos expériences numériques. Nous verrons que notre hypothèse de départ est trop restrictive mais qu'il existe un compromis toutefois très intéressant qui rend possible l'apprentissage avec l'arithmétique à faible précision. Nous parlerons ensuite des circuits que nous avons réalisés sur le processeur V.I.P pour la passe avant et l'apprentissage. Finalement, nous discuterons de considérations logicielles.

5.1 Apprentissage en précision réduite.

Au chapitre précédant, nous avons présenté les équations (1) et (2) avec lesquelles nous comptions faire l'apprentissage et nous avons présenté une esquisse d'une unité de traitement capable de l'apprentissage en matériel. Toutefois, ces équations peuvent être implantées de façons différentes selon ce que l'on peut se permettre en logiciel ou en matériel. La fig. 15 montre les différents types de réseaux et les équations qu'ils utilisent pour l'apprentissage. Le but de ces expériences est de démontrer la viabilité (ou la non-viabilité) des différentes hypothèses liées aux réseaux de neurones à faible précision. Il nous fallait comparer les méthodes afin de déterminer si les suppositions que nous avions faites sur les algorithmes réalisables en matériel à peu de frais sont réalistes et dans quelle mesure.

On se rappellera qu'on a proposé d'utiliser des activations réduites à des puissances négatives de deux, des poids à point fixe de faible précision ainsi que des gradients réduits à des puissances de deux

pour réaliser en matériel un circuit rendant possible l'apprentissage. Rappelons que l'hypothèse à vérifier comporte deux volets et se résume ainsi:

> Pour une architecture de réseau de neurones artificiels donnée, il est équivalent à la méthode traditionnelle d'utiliser des poids de faible précision, par exemple sur 16 bits, et des activations exprimées en puissances de deux pour la passe avant. Si nous avons aussi l'apprentissage à faire, il est probable que d'exprimer le gradient en puissance de deux dans certaines des étapes du calcul soit suffisant.

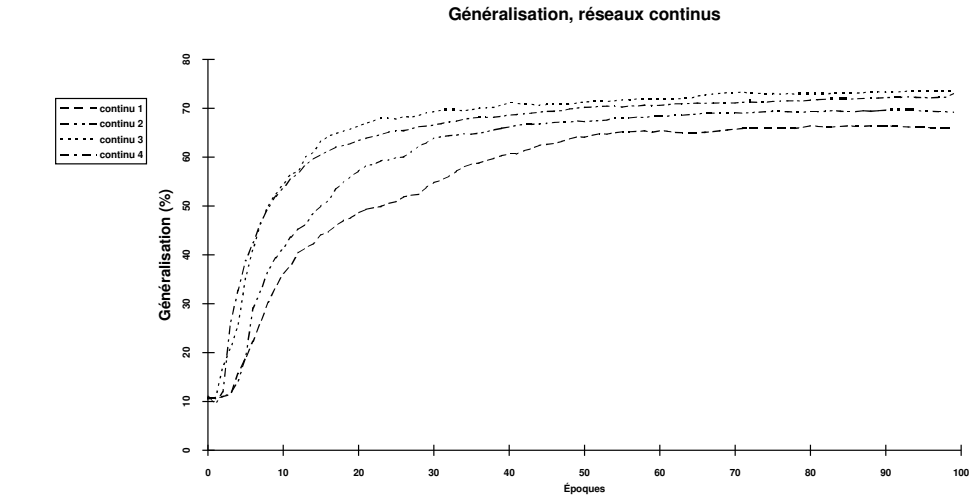
Pour comparer les différents modèles, à savoir « continu », « semidiscret » et « discret », nous avons réalisé des réseaux Lenet 1a, que l'on a entraînés séparément, avec les différentes méthodes sur un même fragment de la base NIST (voir annexe 1). Le but de l'exercice étant uniquement de comparer les méthodes entre elles, nous n'avons pas cru nécessaire de faire les tests avec toute la base NIST, choix d'autant plus justifié que le temps de calcul est très élevé lorsqu'on utilise tous les exemples (plusieurs semaines par réseau pour cent époques). Nous avons utilisé 10,000 exemples pour l'apprentissage et 1000 exemples pour vérifier l'erreur de généralisation (qui pour nous est la performance de la reconnaissance sur des exemples non utilisés pour l'apprentissage). Ce nombre relativement petit d'exemples explique pourquoi la performance plafonne vers 75% plutôt que vers environ 95%. Des tests plus longs ont cependant montré que la version continue était capable d'apprendre aussi bien que la version de Le Cun et al.

	$g_i =$	$\Delta w_{ij} =$	
« semidiscret »	$\tilde{\sigma}'(\tilde{s}_i)\sum w_{ij} g_j$	$-\eta \tilde{x}_i g_j$	$\tilde{s}_i$, w_{ij} et g_j sur 16 bits.
« continu »	$\sigma'(s_i)\sum w_{ij} g_j$	$-\eta x_i g_j$	s_i , w_{ij} et g_j en `float`.
« discret »	$\tilde{\sigma}'(\tilde{s}_i)\sum w_{ij} \tilde{g}_j$	$-\tilde{\eta} \tilde{x}_i g_j$	$\tilde{s}_i$ et w_{ij} sur 16 bits.

Fig. 15. Les différents types de réseaux et les équations et les formats qu'ils utilisent pour l'apprentissage.

Selon la méthode de calcul, différentes expériences ont été réalisées par l'auteur avec des paramètres initiaux différents (graine aléatoire, pas de gradient maximum et minimum, etc.). La fig. 16. montre les résultats obtenus sur une même échelle. Mentionnons que seules les étapes de type convolution sont calculées en faible précision. L'étape pleinement connexe de la fin est toujours calculée en précision élevée (puisque ne correspondant pas au modèle de connectivité à deux dimension du processeur systolique).

On voit que les réseaux continus semblent très stables alors que les semidiscrets oscillent assez fortement. L'oscillation visible dans ces autres réseaux est l'effet du bruit lié à la discrétisation et à l'erreur dans le calcul des gradients. En précision élevée, le bruit associé au calcul d'un gradient demeure faible (dans $O(n\ 2^{-23})$ pour un neurone avec n successeurs) alors que le bruit associé au calcul d'un gradient dans un réseau semidiscret demeure quand même plus élevé, soit dans $O(n\ 2^{-15})$. Individuellement, ces erreurs ne sont pas très importantes mais elles s'additionnent rapidement pour donner un bruit qui rend difficile une estimation correcte du gradient. Pour le réseau discret, nous voyons que le calcul du gradient est tellement mauvais qu'il n'y a pas d'apprentissage. On remarquera que la performance de 10% du réseau discret correspond en fait aux chances de deviner de façon complètement aléatoire la classe d'un exemple.



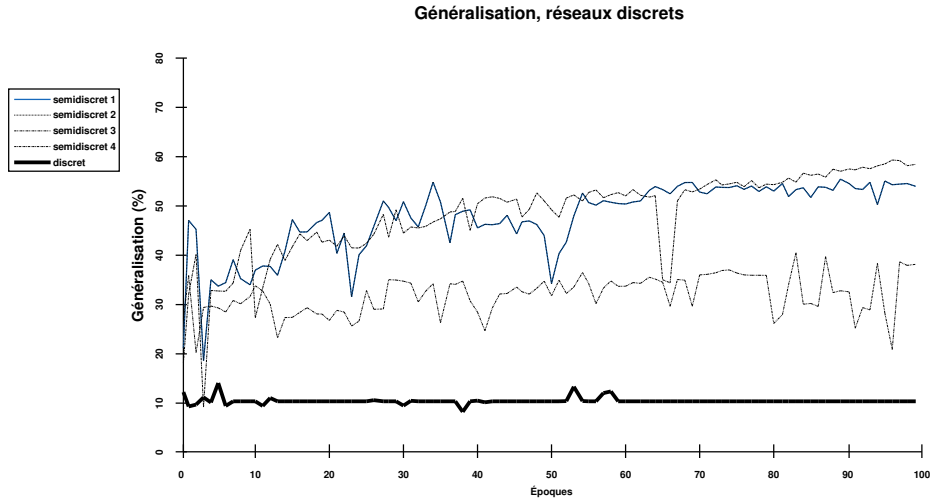


Fig. 16. Généralisation des différents réseaux.

Ces résultats montrent que l'hypothèse selon laquelle le calcul des gradients peut utiliser des gradients représentés par des puissances de deux est *fausse*. Nous pouvons utiliser moins de bits pour les gradients (par exemple 16) mais les représenter en puissances de deux ne permet pas à l'apprentissage de converger. On voit, dans la fig. 16., que les réseaux semidiscrets (ceux qui utilisent une dérivée discrète et des activations représentées en puissances de deux) sont capables d'atteindre, quoique lentement, des

performances comparables aux réseaux continus, selon les paramètres choisis — principalement le pas de gradient pour lequel un choix judicieux est nécessaire. Mais les réseaux discrets semblent incapables d'apprendre.

Les résultats médiocres des réseaux discrets s'expliquent cependant facilement lorsqu'on réalise que les erreurs liées à la discrétisation des gradients ne sont pas indépendantes puisque les gradients sont calculés récursivement. Ainsi, chaque erreur introduite dans le calcul d'un gradient à un neurone *j* se propage à tous les neurones qui le précèdent. Utilisons encore g_i, le gradient exact au neurone *i*, obtenu par un calcul en précision élevée et $\tilde{g}_i$, le gradient obtenu par un calcul en faible précision au neurone *i*. Notons que pour ce qui suit, $\tilde{g}_i$ peut être très différent de $\delta(g_i)$ puisque tout le calcul nécessaire pour produire $\tilde{g}_i$ a été réalisé en faible précision parallèlement à tout le calcul en précision élevée nécessaire pour produire g_i. Le calcul de l'erreur dépendra non seulement de $\delta(\cdot)$, mais aussi de la topologie du réseau. Posons maintenant l'erreur sur le gradient d'un neurone *i* comme étant

$$D(g_i) = (g_i - \tilde{g}_i)^2,$$

ce qui n'est que l'erreur quadratique. Insistons sur le fait que

$$D(g_i) \quad \neq \quad \left(g_i - \delta(g_i)\right)^2$$

Puisque $D(g_i)$ estime correctement l'erreur de $\tilde{g}_i$, on peut calculer l'erreur quadratique moyenne simplement par:

$$D_\mu = E\left[D(g)\right] = E\left[\left(g - \tilde{g}\right)^2\right]$$

où $E[y]$ est l'espérance de y. Utilisons l'estimateur suivant pour calculer D_μ:

$$\hat{D}_\mu = \frac{1}{N}\sum_{i=0}^{N-1}(g_i - \tilde{g}_i)^2$$

où N est le nombre de neurones. Pour estimer l'erreur moyenne expérimentalement, nous avons échantillonné les gradients d'un réseau discret (voir fig. 17). Nous avons obtenu environ 670 millions de gradients (provenant de plusieurs initialisations différentes). L'erreur moyenne trouvée est:

$$\hat{D}_\mu \approx 0.21$$

La valeur exacte de l'erreur moyenne dépend non seulement de $\delta(\cdot)$ mais aussi des valeurs d'initialisation du réseau qui déterminent la distribution initiale des gradients. Différentes expériences mèneront à différentes valeurs pour D_μ mais une erreur du même ordre de grandeur que celle trouvée explique de façon tout à fait convaincante les problèmes observés dans les réseaux discrets. Il s'agira donc de trouver une autre fonction $\delta(\cdot)$ qui minimise D_μ mais qui est encore facile à réaliser en matériel. Il pourrait par exemple s'agir d'exprimer les gradients sous la forme $(-1)^s m 2^e$, pour un bit de signe s, une mantisse m et un exposant e. D'autres travaux seront nécessaires pour déterminer cette fonction.

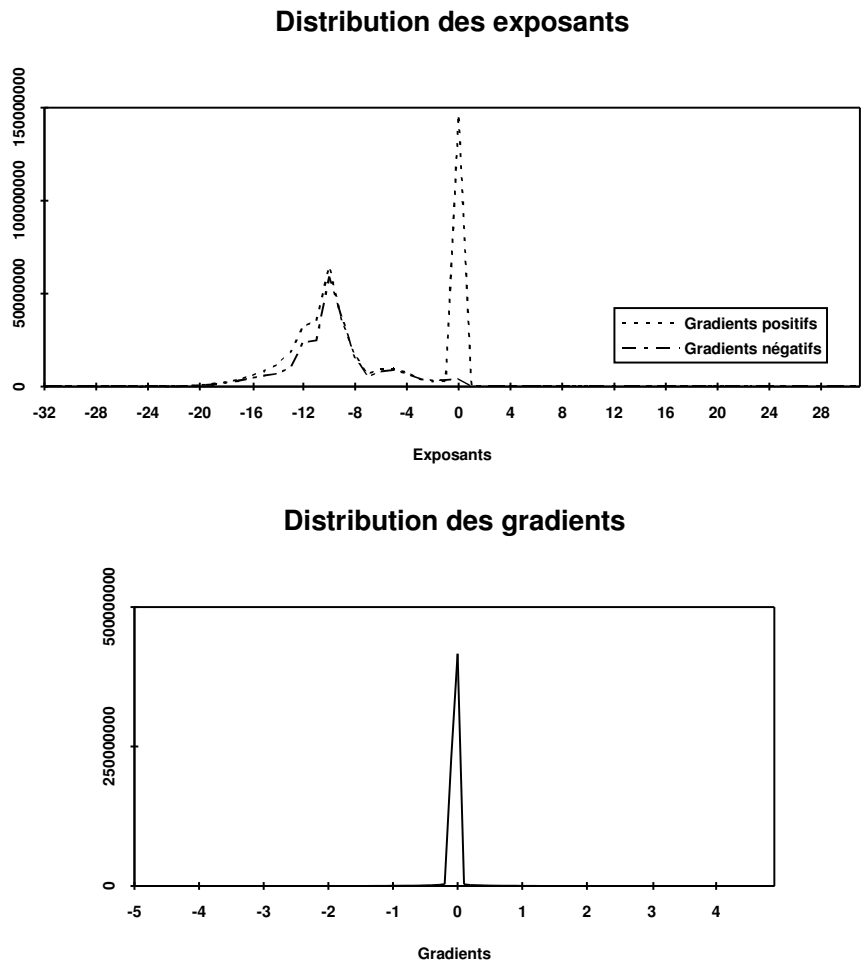


Fig. 17. Distribution des gradients et des exposants.

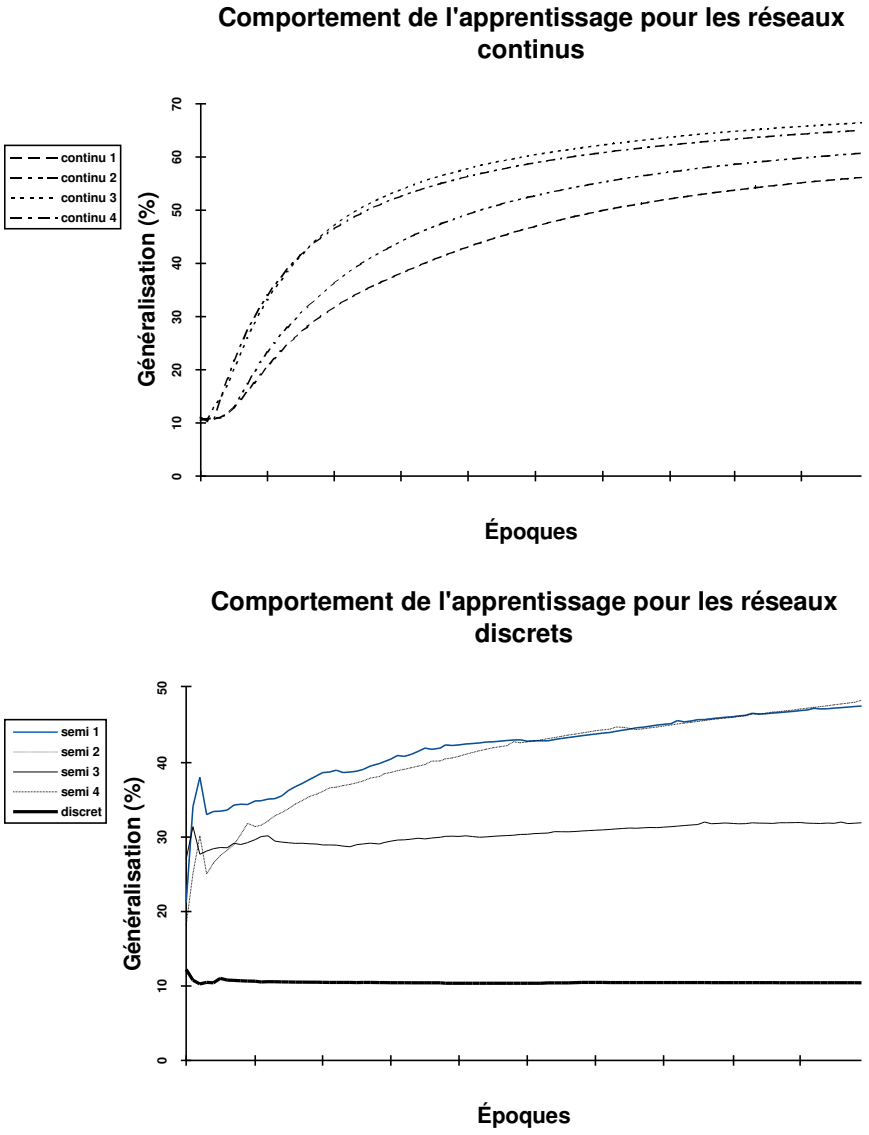


Fig. 18. Comportement de l'apprentissage (moyennes lissées).

La différence entre le meilleur réseau continu et le meilleur réseau semidiscret est d'environ 15%, ce qui ne va pas à l'encontre de nos attentes. On savait que l'introduction de bruit lié à la discrétisation causerait une légère baisse des performances. Nous avions estimé a priori ce déficit à 10%. Lorsque nous examinons les courbes d'apprentissage (fig. 18) nous pouvons voir que les courbes appartenant à un même groupe ont toutes la même apparence. Les réseaux continus semblent tous converger vers une même valeur limite. Il en va de même de trois des quatre réseaux semidiscrets. Si l'apprentissage avait compris plus d'époques, toutes les courbes d'un groupe finiraient éventuellement par se confondre. Cette valeur limite est liée à la capacité du réseau qui est définie par la performance du réseau après un apprentissage avec un nombre infini d'exemple (ou seulement *tous* les exemples possibles). La valeur limite des réseaux

semidiscrets sera nécessairement inférieure à la valeur limite des réseaux continus parce que leur capacité est moindre. Comme la capacité mesure indirectement la quantité d'information contenue dans l'état d'un réseau (c'est à dire ses poids) il n'est pas surprenant de voir que la capacité des semidiscrets est moindre puisqu'il nous faut moins de bits pour exprimer leurs états. Chaque poids dans un réseau continu occupe 32 bits alors que les poids d'un réseau à faible précision n'en ont que 16. Réduire de moitié le nombre de bits semble ne réduire la capacité que d'environ 20%, du moins dans le cas du Lenet 1a.

Ce résultat est surprenant. Dans un chapitre précédant, nous avons discuté plutôt informellement de la stabilité numérique et de la précision des calculs en fonction du nombre de bits que nous utilisions pour les poids. Nous avions conclu que, bien que nous réduisions d'un facteur 2^8 la précision, la précision demeurait suffisante en pratique. Ce qui est moins intuitif, c'est le constat que chaque bit disparu divise par deux le nombre d'états admissibles pour notre réseau de neurones mais que la capacité n'est pas divisée par deux. La précision et la capacité réelles du réseau sont difficiles à calculer. À chaque neurone s'insère une certaine quantité de bruit qui est plutôt compliquée à caractériser. C'est quand même ce bruit qui limite de façon pratique le nombre de bits significatifs. La précision d'un poids dépendra du nombre de bits qui ne seront pas bruités. Les différences dans les performances suggèrent qu'en fait le nombre de bits *précis* ne diffèrent pas vraiment d'un système à l'autre (puisque le ratio des capacités est environ 1.2 : 1). Pour balancer la perte de capacité liée à la réduction de la précision, nous pouvons augmenter le nombre de poids ou modifier l'architecture de notre réseau. Dans le cas d'un réseau LeNet 1a, par exemple, il serait peut-être adéquat de rajouter quelques cartes.

5.2 Réalisations avec le processeur V.I.P

La réalisation matérielle des UT pour la passe avant était aussi un objectif important. Elle met en évidence les principales difficultés que l'on doit s'attendre de rencontrer lorsqu'on fait de la conception au niveau matériel. Nous avons programmé les UT de la fig. 13. et des unités réduites (UR) qui ne contenaient que le registre R et R_t. Les UR ne servent qu'à déplacer l'information sur les rebords de la matrice. Leur grande simplicité permet qu'on en mette un grand nombre sur un seul circuit à logique programmable. La fig. 19. montre la configuration finale.

Les UT et les UR communiquent de voisins à voisins selon la grille torique déjà présentée. Pour permettre le routage avec les outils Altera, nous avons dû multiplexer le décaleur/additionneur dans l'UT et pipeliner la fonction sigmoïde. Le résultat global étant que les poids sont diffusés en deux temps (la partie basse puis la partie haute) et qu'il faut un cycle oisif après le dernier calcul sur Σ pour pouvoir copier le résultat de *f* dans R.

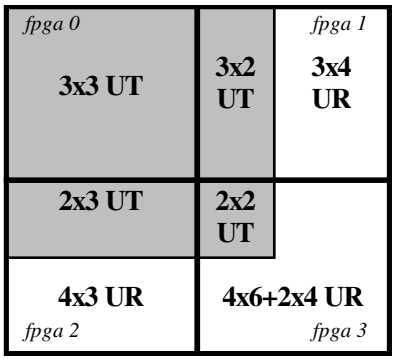


Fig. 19. Répartition des UT et des UR sur la matrice systolique.

Avec cette configuration matérielle, nous pouvons exécuter approximativement 116000 x 25 convolutions 5x5 par seconde. On compte 138 cycles pour un groupe de 25 convolutions (avec le chargement et la récupération des données). La cadence du processeur V.I.P. est 16Mhz. Ces 2,898,550 convolutions correspondent à peu près à traiter une image de 1700^2 pixels en une seconde. Pour une image de 32x32, le temps serait donc d'environ 0.3ms. Les temps obtenus et présentés dans [Cl95] dépendaient de UT plus simples (1 bit par pixel et 2 bits par poids) de type NET32K et d'une plus grande vitesse d'horloge. Dans notre contexte, les poids sont de 16 bits et les pixels sont de 3 bits. Le nombre de UT de type LeNet est par conséquent très inférieur au nombre de UT de type NET32K que l'on peut faire entrer sur nos FPGAs. On peut donc considérer que réaliser 25 convolutions 5x5 en 8.6µs est très satisfaisant.

Pour les UT capables d'apprentissage, la situation est beaucoup moins avantageuse. Nous n'avons pu programmer que deux de ces UT (voir fig. 14) par circuit (pour un total de 8) et au coût de tout multiplexer ou presque. Dans ce cas, il est clair que ce n'est pas le nombre de portes logiques qui fait défaut mais le nombre de connexions (dû à l'architecture Altera). Puisque nous avons dû même multiplexer les connexions entre les voisins, nous divisions par 2 la performance. Si nous n'avons qu'un seul UT par circuit avec des communications non multiplexées nous avons la même performance. Finalement, nous n'avons que la performance équivalant à 4 UT, ce qui est de beaucoup inférieur au minimum suggéré au chapitre précédent, soit environ 16.

Pire, ces UT pour l'apprentissage utilisent un gradient discrétisé à une puissance de deux. Nous avons montré que cette méthode n'était très pas bonne. Il faudra utiliser des UT dont le gradient est un nombre à point fixe (sur 16 bits par exemple). Avec les circuits Altera que nous utilisons, nous serons définitivement contraints à n'avoir que 4 UT pour l'apprentissage en matériel.

5.3 Réalisations en logiciel

Nous avons dû aussi réaliser des outils logiciels pour supporter l'environnement V.I.P. Le simulateur est de loin le plus complexe de ceux-ci. Puisque nous en avons discuté suffisamment dans un chapitre précédent, nous ne reviendrons pas sur ses caractéristiques. Vient en suite l'assembleur de microcode, SAS, qui, sans être aussi complexe que le simulateur est tout de même un outil puissant puisqu'il est aisément reconfigurable.

Du côté des simulations, nous avons trois programmes qui ont des caractéristiques d'exécution différentes. Bien que l'on puisse penser qu'un programme basé sur l'arithmétique entière soit plus rapide qu'un programme basé sur l'arithmétique à virgule flottante, c'est la version continue qui est de beaucoup la plus rapide. Cela tient principalement au fait que la fonction d'activation $\tilde{\sigma}_i(s_i)$ et sa dérivée $\tilde{\sigma}'_i(s_i)$ ne sont pas aussi faciles à calculer de façon efficace qu'on pourrait le croire. Dans la version où nous utilisons $\tilde{g}_j$, nous devons aussi calculer la puissance de deux la plus près de g_j, ce que nous calculons lorsque nous appliquons $\delta(g_j)$. Si nous n'utilisons pas une table pour calculer $\tilde{\sigma}_i(s_i)$ (et $\delta(g_j)$), nous avons au mieux dans O(log *n*) branchements pour *n* partitions de l'espace d'entrée. Nous avons aussi observé que le temps d'exécution des réseaux semidiscrets et discrets variaient grandement selon si la machine qui exécutait le programme avait ou non une unité de prédiction des sauts conditionnels. Si nous voulions utiliser une table pour calculer $\tilde{\sigma}_i(s_i)$, il nous faudrait autant d'entrées que de valeurs admissibles pour s_i. Si cette technique est réalisable pour un prototype, elle ne l'est guère pour les applications pratiques qui sont souvent sévèrement contraintes côté mémoire. Il faut préciser que calculer la fonction $\tilde{\sigma}_i(s_i)$ de façon combinatoire mène aussi à beaucoup de manipulations au niveau des bits, ce qui n'est en général pas très efficace sur les processeurs généraux. De plus, les multiplications du type $w_{ij}\tilde{x}_j$ ne peuvent se calculer directement par $w_{ij} >> \tilde{x}_j$; il faut aussi faire un test pour ajuster le signe du résultat selon la valeur de $\tilde{x}_j$. Ces opérations sont rarement efficaces lorsqu'on écrit le programme dans un langage de haut niveau.

Nous avons aussi montré que les algorithmes de convolution rapide ne peuvent nous être utiles que dans certains cas très limités. En général, toutefois, et dans le cas qui nous intéresse, la nature apparemment aléatoire des noyaux empêche tout algorithme de vraiment optimiser le calcul. Le premier algorithme décrit dans la section 3.2. peut ramener le calcul d'une convolution en O(*n* log *n*) pour *n* pixels si les noyaux sont adéquats. On pourrait prendre cet algorithme et les noyaux de la transformée discrète de Fourier et on retrouverait à la sortie une version (probablement méconnaissable) de la transformé discrète rapide de Fourier. Le second algorithme est plutôt une astuce qu'un algorithme à proprement parler. Il s'agissait de précalculer les sommes partielles et de les stocker dans une table que l'on consultait par la

suite. Comme on pouvait précalculer des sommes partielles de 8 ou 16 termes on obtenait un gain intéressant, mais avec des contraintes sévères (un bit par pixel et très peu de bits par poids). Ces algorithmes demeurent intéressants même s'il s'agit de cas spéciaux.

Chapitre 6: Conclusion.

Nous discuterons des impacts des résultats obtenus et des possibilités d'amélioration. Certains résultats obtenus sont indépendants de la technologie alors que d'autres en dépendent fortement. Nous discuterons de chacun d'eux. Rappelons ces résultats importants:

- La passe avant peut être réalisée avec des activations réduites à des puissances de deux et des poids de faible précision avec des résultats satisfaisants.

- L'apprentissage est possible mais seulement lorsque les gradients sont toujours représentés avec une certaine précision. L'utilisation de puissances de deux pour les gradients ne semble pas permettre que l'apprentissage converge, contrairement à ce qui avait été spéculé.

- La réalisation matérielle d'un processeur systolique sur circuits à logique programmable pour la passe avant est possible et donne de bons résultats, même avec une vitesse d'horloge assez basse et un petit nombre de UT.

- La réalisation matérielle d'un processeur systolique pour la passe avant et l'apprentissage n'est guère pratique avec les circuits à logique programmable actuels, en particulier ceux qui sont utilisés sur la carte V.I.P. Même si la réalisation de ce processeur est possible, nous n'atteignons pas le seuil de rentabilité. L'apprentissage demeure pour l'instant plus rapide en logiciel, mais l'amélioration de la technologie offrira probablement des solutions intéressantes à court terme.

Nous savons que la passe avant ne souffre que très peu du passage à l'arithmétique à faible précision, pourvu, toutefois, que les poids soient ajustés à ce mode d'évaluation. Si on prend un réseau continu préalablement entraîné et qu'on le discrétise, il faudra faire un entraînement supplémentaire de

quelques époques pour compenser pour l'erreur introduite avec la discrétisation. Cette technique a été employée à maintes reprises lors d'expériences conduites par de tierces parties.

Nous avons montré que l'apprentissage avec des poids de précision réduite et des activations exprimées en puissance de deux était réalisable mais pas si on utilisait aussi des gradients sous forme de puissance de deux. Le problème provenant principalement de l'accumulation succesive des erreurs sur les gradients. D'ailleurs, de nombreux auteurs utilisent une précision quand même assez élevée pour leurs gradients. Lyon et al [Ly96] utilisent 8 bits pour les gradients plus 16 bits supplémentaires pour le contrôle de l'erreur — finalement leurs gradients sont sur 24 bits. Aihara et al [Ki96] utilisent aussi des gradients sur 16 bits. Nous utilisons des gradients sur 16 bits. Les résultats obtenus ne sont cependant pas exaltants. L'apprentissage est possible, mais avec une convergence moins rapide et plus bruitée. La capacité totale d'un réseau est moindre lorsqu'il est discrétisé et cela explique la différence de performance observée. Il faut donc compenser cette perte de capacité par une augmentation du nombre de poids.

Les performances obtenues avec la réalisation matérielle d'un processeur systolique calculant la convolution sur le processeur V.I.P. sont très encourageantes lorsqu'on considère le coût de conception et de fabrication du processeur V.I.P., le temps mis à la réalisation des UT et d'autres facteurs comme la vitesse d'horloge et le nombre de cellules logiques disponibles. Calculer une convolution 5x5 sur une image de 32x32 en 0.3ms est quand même très intéressant. Si le besoin s'en faisait vraiment sentir, il serait possible de réaliser le processeur systolique en VLSI pour disposer de vitesses d'horloge élevées et d'un plus grand nombre de UT.

Si nous montions cette hypothétique puce VLSI sur une carte d'extension, nous serions toujours aux prises avec les problèmes qui surviennent lorsqu'on a un lien (relativement) lent entre deux processeurs. Toutefois, certains processeurs supportent des *coprocesseurs*. Ces processeurs communiquent avec leurs différents coprocesseurs par un bus local qui permet d'envoyer efficacement instructions et données. Par exemple la famille 80x86 d'Intel supporte ce type de canaux privilégiés. Ces canaux ont d'abord été pensés afin d'ajouter un coprocesseur pour l'arithmétique en point flottant. La technologie du début des années '80 rendait impossible ou très difficile la réalisation d'un processeur incorporant aussi l'unité arithmétique en point flottant; il était obligatoire soit de l'abandonner, soit de le réaliser sur une puce distincte. Si nous optons pour la réalisation sur une puce distincte, nous devons aussi penser à un protocole de communication efficace entre les deux parties. Puisque les opérations réalisées par le processeur et le coprocesseur s'exécutent en parallèle, il serait inefficace de faire attendre le processeur principal; il faut que le protocole de communication permette de fonctionner par *demandes* et de façon asynchrone. On fait une demande au coprocesseur, en lui passant instructions et données, puis on attend

que celui-ci nous dise qu'il a terminé son calcul mais nous pouvons continuer l'exécution du programme principal entre-temps.

En fait, ce qui arrive, c'est que nous transférons le contrôle du bus local (reliant processeurs et mémoire) au coprocesseur le temps qu'il obtienne les données dont il a besoin, puis celui-ci retourne le contrôle au processeur principal dès que les calculs commencent. Tant que le coprocesseur calcule, le processeur principal est libre de procéder à sa guise. Lorsque le coprocesseur termine ses calculs, il sauvegarde les résultats dans sa mémoire et il informe le processeur qu'il a terminé par l'activation d'un signal dédié. Ce dernier devra récupérer les résultats [Hu92]. Cette méthode nous permettrait d'atteindre des performances vraiment intéressante tout en disposant d'un environnement d'ordinateur général.

Mais à tous ces avantages vient s'opposer un problème de taille. Si par le passé presque toutes les cartes mères offraient une fiche pour un coprocesseur numérique, les cartes mères courantes profitent du fait que, depuis l'avènement du 80486DX, le coprocesseur numérique est sur la puce même. Ce qui veut dire que, si l'on voulait utiliser un Pentium comme processeur principal, il faudrait vraisemblablement aussi concevoir une carte mère afin d'accommoder le coprocesseur, ou un module de type overdrive… ce qui n'est pas nécessairement intéressant.

Mais il se pourrait aussi que la seule possibilité pour l'apprentissage en matériel soit, à court terme, des puces VLSI comprenant un assez grand nombre de UT. Nous avons montré qu'il était possible d'apprendre en matériel, mais qu'il fallait toutefois un assez grand nombre de UT pour dépasser la performance atteinte en logiciel avec un processeur général. Un pentium à 100Mhz peut probablement faire de 20 à 25 millions de multiplication-additions par seconde et évaluer la dérivée d'une activation en une centaine de cycles ou moins. Les performances seront dix ou vingt fois plus grandes si on utilise un processeur DEC Alpha à 500 Mhz. Ce sont ces performances qui déterminent le seuil de rentabilité de l'apprentissage en matériel, pour le secteur de la recherche du moins. Si le matériel spécialisé ne peut dépasser ces performances, il doit être abandonné. Quant aux applications pratiques, on a en général un réseau déjà entraîné qui doit s'adapter un peu dans le temps. Dans ce cas, la performance n'aura pas besoin d'être aussi grande mais elle doit quand même demeurer supérieure à ce que l'on pourrait faire avec un simple microcontrôleur. Le seuil de rentabilité est alors défini par les besoins en termes de puissance de calcul et par l'argent que l'on est prêt à y mettre.

Pour l'instant, la carte V.I.P n'offre pas suffisamment de puissance de calcul pour que l'apprentissage soit intéressant en matériel, mais c'est déjà suffisant de montrer que c'est possible et qu'éventuellement des processeurs de ce type pourraient être produits à relativement peu de frais et incorporés dans de nombreuses applications.

* * *

Finalement, nous avons rejoint nos principaux objectifs. D'abord, la vérification des différentes hypothèses sur l'apprentissage par le biais d'expériences logicielles. Nous savons qu'il est possible d'apprendre pourvu que les gradients ne soient pas des puissances de deux et nous avons entrevu quel était le lien entre la précision et la capacité. Ensuite, nous avons vu que s'il était rentable d'utiliser un ordinateur à logique programmable pour un processeur systolique calculant des convolutions (essentiellement la « passe avant ») il n'était cependant pas encore rentable de l'utiliser pour l'apprentissage. Ce manque de rentabilité n'est pas lié au concept mais seulement à la capacité de l'ordinateur à logique programme utilisé, le processeur V.I.P. Il est certain que le futur nous apportera des circuits à logique programmable offrant beaucoup plus de ressources et qu'à ce moment l'apparition d'ordinateurs à logique programmable de plus grande capacité rendra l'apprentissage (et beaucoup d'autres applications!) en matériel très intéressant.

ANNEXE 1: réseaux multicouches de type LeNet et la base NIST

Nous avons déjà parlé de l'application cible, mais dans cette annexe nous nous attarderons longuement sur l'architecture du réseau LeNet 1a. Nous décrirons en détail les différentes cartes. Comme il s'agit d'une application de reconnaissance de caractères manuscrits, nous devions disposer d'une banque d'exemple. Tous les exemples proviennent de la base du NIST (le *National Institute of Standards and Technology* américain) qui contient 120 000 caractères numérisés, que l'on suppose correctement étiquetés. Ces caractères sont encodés dans une matrice de 28x28 pixels de 256 niveaux chacun, le niveau 255 représentant l'encre et le 0 le papier. Cette base de données est divisée en deux volets; les premiers 60000 caractères sont considérés lisibles et faciles; alors que les derniers 60000 sont considérés difficiles. La fig. 1. en montre quelques-uns provenant de la première partie. Les chiffres de la base sont raisonnablement distribués et nous n'avons pu constater qu'un très faible biais (tous les chiffres sont en même nombre, à quelques pour cent près) alors nous pouvons faire abstraction de ce détail pour l'apprentissage. S'il en avait été autrement, ç'aurait été un sérieux problème! De plus, ils sont déjà randomisés, ce qui nous évite d'apprendre dans une seule direction à la fois.

Or, la tâche d'un réseau LeNet est justement d'identifier ces caractères sans erreur, pour que l'on puisse éventuellement inclure des mécanismes de lecture automatique dans certaines applications comme des trieuses de courrier automatiques ou des systèmes qui permettraient de lire des chèques et ainsi de diminuer le délai de traitement bancaire. Le réseau LeNet est un réseau relativement simple mais qui donne de très bons résultats. Cela est dû principalement aux cartes qui jouent un rôle très important dans les problèmes de reconnaissance spatiale qui, lorsque superposées, mènent à un effet d'amplification de

l'information. Cette information amplifiée est par la suite utilisée par un réseau de neurones classique qui dispose d'une information filtrée pour calculer sa sortie.

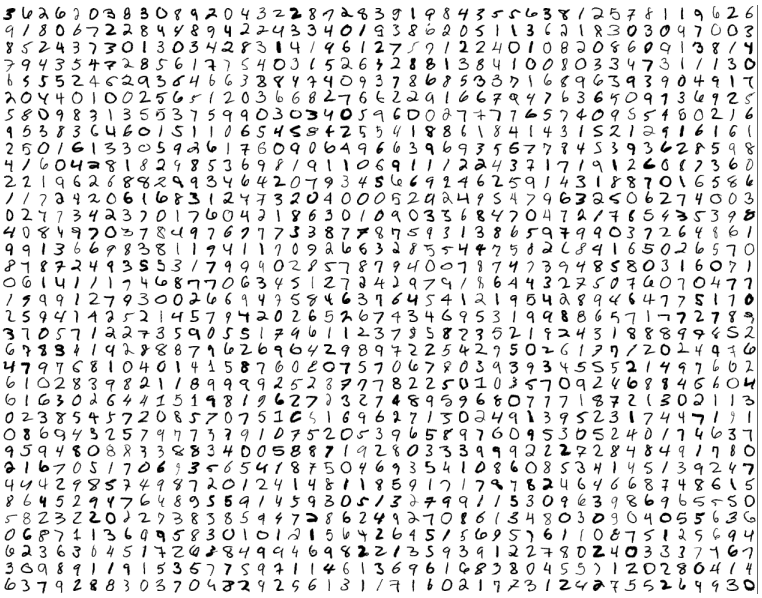


Fig. 1. Un segment de la base de données du NIST.

Anatomie d'un réseau LeNet.

Le lecteur voudra considérer la fig. 2. afin de mieux suivre nos propos. L'image NIST est agrandie à 32x32 pixels en entrée puis on lui applique quatre noyaux différents qui forment les quatre premières cartes de 28x28 pixels. Chacune de ces cartes est sous échantillonnée avec un noyau différent pour donner une carte de 14x14 pixels. De nouveau, des convolutions sont appliquées sur ces cartes. Les convolutions de ce niveau, toutefois, sont légèrement différentes. Chaque pixel des cartes de l'étage suivant est en fait la somme de deux convolutions qui se partagent le même biais:

$$x_j = \sigma_c\left(b_j + \sum_{(a,b)\in pred(j|A)} A_{a,b} x_{a,b} + \sum_{(a,b)\in pred(j|B)} B_{a,b} x_{a,b} \right)$$

Où A et B représentent les deux noyaux associés aux images sources (indiqués par un pointillé dans la fig. 6.). Les cartes résultantes sont de 10x10 pixels. On poursuit ensuite par un sous échantillonnage qui donne des cartes de 5x5. Finalement, on procède à l'évaluation pleinement connexe, où les 12x5x5 neurones des cartes sont branchées à chacun des 10 neurones de sorties.

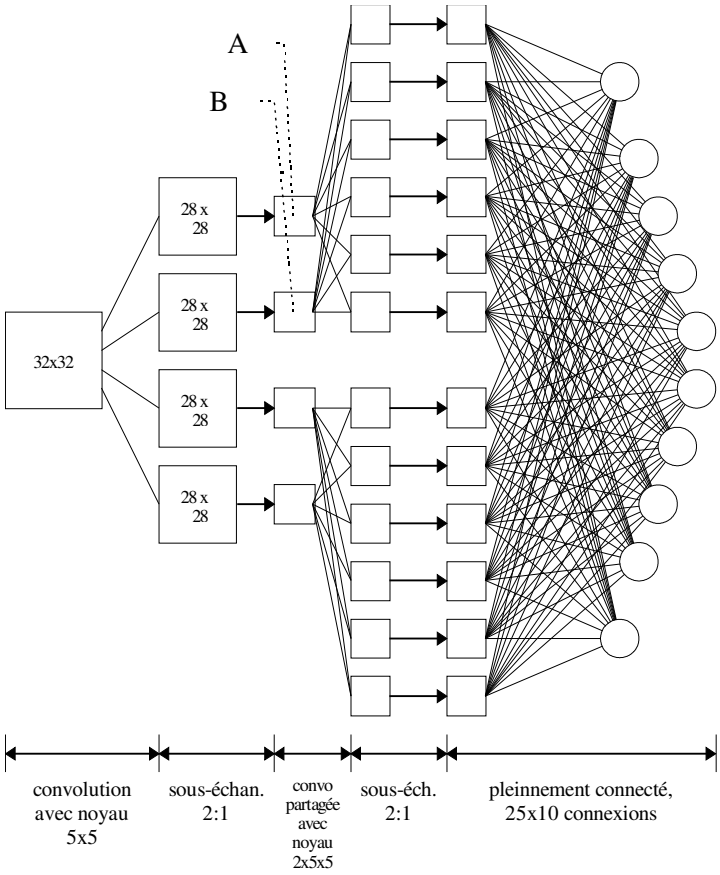


Fig. 2. L'architecture d'un réseau LeNet 1a. Les lignes fléchées indiquent les étapes de sous échantillonnage où s'opére une dépopulation des pixels par moyennage (le biais et la méthode de moyenne sont appris par le réseau) tandis que les lignes simples indiquent des convolutions. Dans la couche pleinement connexe, chaque ligne remplace les 25 synapses qui relient chaque neurone des cartes à un neurone de sortie.

ANNEXE 2: manuel d'instruction de SAS

Cette annexe reproduit verbatim le manuel d'instruction écrit pour l'assembleur systolique SAS v1.0. Comme le manuel original a été écrit pendant notre séjour de l'été 1995 à Bell Labs, le lecteur ne doit pas se surprendre de le voir écrit en en anglais.

SAS "CONVOLUTIONS"

parallel assembler for systolic processor
assembly language reference manual.

1. SAS program structure and syntax requirements

SAS programs has two main clauses. The first, the **defmask** clause, is used to define the masks that are to be used in the systolic processor. Masks define which processors accept instructions in the array and which do not. The second clause is the **code** clause, is used to tell what the program will be. Both clauses are ended by the **end** reserved word, and each statement in the clauses are delimited by a semicolon **;** .

1.1 General syntax

The SAS assembler is case insensitive. That is, unlike C, "*Instruction*" is the same as "*INStrucTiOn*" and "*instruction*". SAS also allows spaces. Due to limited number of reserved word and its simple language, SAS let you use spaces in instructions. So, "*clear S*" is the same as "*clears*", and "*adr++*" can be written "*adr + +* ".

Labels are given by a name preceded by the 'at' sign: '@'. A label name may contain every caracter you like, except the following: **;** , **,** ,**\n**, **:** , **\t**, and space(s), which are delimiters.

The **{** and **}** are the comment delimiters. Comments can be nested, althought it might not be a good idea to nest comments more than once or twice for program readability. As comments are eaten away during the parsing, you can also insert a comment inside a keyword. A comment may also span across many lines and contains '**;**' that are usually instruction delimiters.

1.2 The **defmask** clause.

The **defmask** clause is always given by the following regular expression:

defmask;
[<*mask*>;]*
end;

where <*mask*> = { 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,A,B,C,D,E,F}^{64}

Therefore, a mask is a hexadecimal number 64 digit long. In the 8x8x2x2 processor model, the processors are numbered sequentially. That is, the first processor is #0, and receives the bit 0 the mask, and the first processor of the second FPGA is #8, so it receives the bit 8 of the mask. The first processor on FPGA 3 is number 32, and the first processor on the last FPGA is #40. The last processor is #63 and is in FPGA 4. Numbering of processors depends on the current version of the systolic array.

1.3 The **code** clause.

The code clause is always given the the following regular expression:

code;
[[<*label*>]? <*instruction*>[,<*instruction*>]*;]*
end;

where <*label*> = @ {'0' ,..., '9', 'a' ,..., 'z', 'A',...,'Z', '@',...,' * ' }+**:**
<*instruction*> = {
ADR++,
ADR=++COL_ADR,
ADR=++ROW_ADR,
ADR=COL_ADR,
ADR=RESULT_ADR++,
ADR=--ROW_ADR,
ADR=TILE_ADR,
ADR=TILE_ADR+=2,
ADR=TILE_ADR+IMAGE_WIDTH*2,
BREAKPOINT,
CLEARS,ENABLES,SAVES,
COL_ADR=COL_IDX,
COL_IDX+=NB_COL,
CS0000,CS0001,CS0010,CS0011,
CS0100,CS0101,CS0110,CS0111,

```
CS1000,CS1001,CS1010,CS1011,
CS1100,CS1101,CS1110,CS1111,
LOADSIGMA,ENABLESIGMA,
LOOP1,LOOP2,JUMP,CALL,RET,REP,EXIT,
NB_LOOP1=IMAGE_WIDTH-2,
NB_LOOP2=IMAGE_HEIGHT-2,
NB_REP=NB_COL-3,
NOP,LOAD,
ROL,ROU,ROD,
ROW_ADR=ROW_IDX,
ROW_IDX+=NB_ROW,
SHIFT1,SHIFT2,SHIFT3,SHIFT4,SHIFT5,
SHIFT11,SHIFT12,SHIFT13,SHIFT14,SHIFT15,
SHIFT16,SHIFT17,SHIFT18,SHIFT19,SHIFT20,
SHIFT21,SHIFT22,SHIFT23,SHIFT24,SHIFT0,
SHIFT6,SHIFT7,SHIFT8,SHIFT9,SHIFT10,
STORE_COL
}
```

2. Register map.

2.1 The Registers

The systolic controller contains 17 user-accessible registers. They are used for program control. There is two types of registers: read/write registers, that you can read and write from the controller and read and write from the program, and the "read only" or working registers that you can only read from the controller and assign by instructions in the program.

register	**bits**	**mode**	**utility**
IP	16	r/w	program pointer
ADR	16	w	Adress pointer
TILE_ADR	16	r/w	points to the tile to processed next
RESULT_ADR	16	r/w	points to where the result will be stored
IMAGE_WIDTH	16(10)	r/w	image width in tiles
IMAGE_HEIGHT	16(10)	r/w	image height in tiles
NB_COL	8(5)	r/w	number of kernel columns
NB_ROW	8(5)	r/w	number of kernel rows
COL_IDX	16	r/w	Adress of a column in memory
ROW_IDX	16	r/w	Adress of a row in memory
FLAGS	16	r/w	Determines program activity
RP	16	hidden	Keeps the return address
NB_LOOP1	8	w	Keeps count for LOOP1
NB_LOOP2	8	w	Keeps count for LOOP2
NB_REP	8	w	Keeps count for REP
COL_ADR	16	w	Points to a column in memory
ROW_ADR	16	w	Points to a row in memory

Notes: a tile is a 32x32 1bit/pixel image.
NN(nn): NN bits as seen from the outside (from the controler) and nn as seen from the systolic controler. As registers are memory mapped, speed efficient encoding have been made, so that registers are expanded to the next integer number of bytes.

2.2 The flags

The flags register processor modes and activity. There are state flags, and mode flags. State flags show what activity went on in the processor during last cycle(s). The state flags are C3,C2 and C1, which are the borrows from NB_REP, NB_LOOP2 and NB_LOOP1 respectively. The A flag is at 1 if the ADR register was accessed during last cycle, 0 otherwise. The 8 upper bits of the flags (bits 8 to 15) are the same 8 bits found in IP in position 0 to 7. This is used for debugging purposes.

The mode flags are B, the breakpoint enable bit, and M1 and M0, which determined the systolic processor execution mode. if the B bit is a 1, then BREAKPOINT instruction is honored and the systolic controller falls in hold mode. The M1 and M0 decides in which mode the instructions are processed. These modes are:

M1	M0	Mode
0	0	Idle (program completion, after EXIT)
0	1	Hold (program hold or BREAKPOINT)
1	0	Step by step
1	1	Running

The actual RUN instruction is send by the EPLD controller via the standard interface (see this document for more information).

3. The instructions.

On the folling pages are the description of each class of instruction with syntax information, behavior and other information as it applies.

ADR related instructions

Syntax:
```
adr++
adr=++col_adr
adr=++row_adr
adr=col_adr
adr=result_adr++
adr= - - row_adr
adr=tile_adr
adr=tile_adr+=2
adr=tile_adr+image_width*2
```

Those instruction are used to affect the adress to be presently broadcasted on the systolic processor adress bus. This adress affects all memory operations performed by the FPGAs at that time.

Instruction	Effect
ADR++	Increments the address pointer to the next word.
ADR=++COL_ADR	Preincrements COL_ADR then moves it to ADR.
ADR=++ROW_ADR	Preincrements ROW_ADR then moves it to ADR.
ADR=COL_ADR	Copies COL_ADR to ADR.
ADR=RESULT_ADR++	Copies RESULT_ADR to ADR, then postincrements it.
ADR=--ROW_ADR	Predecrements ROW_ADR and copies it to ADR.
ADR=TILE_ADR	Copies TILE_ADR to ADR.
ADR=TILE_ADR+=2	Preadds 2 to TILE_ADR and copies it to ADR.
ADR=TILE_ADR+IMAGE_WIDTH*2	Precalculates effective adress to ADR

BREAKPOINT

Syntax: breakpoint

The BREAKPOINT instruction puts the systolic controller in hold mode if the breakpoint enable bit B is set to 1. Otherwise, the BREAKPOINT instruction is ignored and program execution continues as usual. The conditionnal BREAKPOINT (being enabled by the B bit) allows to write a program with breakpoints and not recompile it once it is debugged.

The BREAKPOINT has no side-effect: all registers are unaffected, and the breakpoint is honored *before* the instruction is completed.

CALL

Syntax: call <label>

where <label> = @ {'0' ,..., '9', 'a' ,..., 'z', 'A',...,'Z', '@',...,' * ' }**+:**

CALL copies IP to RP, and moves the address associated with <label> to IP. As every other jumps, CALL takes effect only one cycle later. Thus, the instruction immediately following the CALL is executed before the actual jump takes place.

CLEARS,
SAVES,
ENABLES

Syntax: clears
saves
enables

The CLEARS instruction loads a 0 into the S register of each PE (according to the current mask).

The SAVES instruction (according to the current mask) copies all active S registers to a memory location pointed by the ADR registers and selected with CSxxx. This is how results are stored in memory.

The ENABLES instruction permits the S register to be loaded with the output of the G function, (according to the current mask).

COL_ADR = COL_IDX

Syntax: col_adr=col_idx

This instruction copies the COL_IDX pointer to the current COL_ADR. COL_IDX is used as an original value for the column pointer, while COL_ADR is the working value. There is no other effect.

COL_IDX+=NB_COL

Syntax: col_idx+=nb_col

This instruction adds the contents of NB_COL to COL_IDX. COL_IDX is used as an original value for the column pointer. However, to pass to the next tile, you must add NB_COL to it.

CSxxxx

Syntax: CS0000
CS0001
...
CS1110
CS1111

The CS0000 through CS1111 instructions enable the different memory channels (chip selects). Each memory banks (one by FPGA) has 4 memory channels, or chip selects. CS0000 selects no memory channels while CS1011, for instance, selects all except the second channel (channel #1, the first being #0)

The CSxxxx instructions are mainly used while preprocessing and post processing images before and after convolutions.

EXIT

Syntax: exit

the EXIT instruction set the systolic controller into the idle mode. No other flags are affected. This instruction is the normal program termination. The instructions following EXIT is not executed, since all systolic controller functions are suspended.

LOAD

Syntax: load

LOAD loads the R register (according to the current mask) with a bit from the memory location pointed by ADR and selected with CSxxxx and SHIFTxx. It does not affect any flags. This is how information is loaded from memory to the PE.

LOADSIGMA,
ENABLESIGMA

Syntax: loadsigma
enablesigma

The LOADSIGMA instruction copies the value being broadcasted at that time into the Sigma register of each PE (according to the current mask).

The ENABLESIGMA instruction enable the adder attached to the Sigma register in each PE (according to the current mask). It then adds the old value of sigma to the value outputed by the F function.

LOOP1,
LOOP2

Syntax: loop1 <label>
loop2 <label>

where <label> = @ {'0' ,..., '9', 'a' ,..., 'z', 'A',...,'Z', '@',...,' * ' }+**:**

LOOPx jumps if NB_LOOPx is greater than or equal to zero. Since jumps are postponed one cycle, when LOOPx branches the following instruction is being executed. If LOOPx does not loop, program execution proceed to the next instruction. LOOPx affects the C2 and C1 flags.

JUMP

Syntax: jump <label>

where <label> = @ {'0' ,..., '9', 'a' ,..., 'z', 'A',...,'Z', '@',...,' * ' }+

JUMP loads IP with the adress pointed by <label>. The JUMP takes effect during the next instruction, so that the instruction immediately following the JUMP is always executed.

NB_LOOP1 = IMAGE_WIDHT-2,
NB_LOOP2 = IMAGE_HEIGHT-2

Syntax: nb_loop1=image_width-2
nb_loop2=image_height-2

These instructions load the loop counters with predefined values (stored in image_width and image_height). NB_LOOP1 = IMAGE_WIDTH-2 resets C1 to 0, and NB_LOOP2 = IMAGE_HEIGHT-2 resets c2 to 0.

NB_REP = NB_COL-3

Syntax: nb_rep=nb_col-3

This instruction moves nb_col-3 to nb_rep, the repeat loop counter, leaving unaffected nb_col. This instruction clears the C3 (setting it to 0).

NOP

Syntax: nop

Nop performs no instruction (other than incrementing IP to the next instruction). It does not affect any flags and lasts a cycle.

REP

Syntax: rep

The REP instruction prevents the IP from change to the next instruction, as long as NB_REP is greater than or equal to zero. The other instruction is still prefetched but is not executed. It affects the C3 flag. If NB_REP is negative at that time, REP wont repeat the instruction. You will have to reset the C3 before doing another REP.

RET

Syntax: RET

RET copies RP to IP. As every other jumps, RET takes effect only one cycle later. Thus, the instruction immediately following the RET is executed before the actual jump takes place.

ROL,
ROU,
ROD

Syntax: rol
rou
rod

These instructions determine the rotation of information via the R registers of all PEs. The rotate "nop" is LOAD, which can be disabled using the current mask. If a PE is not selected by the current mask, its R register do not memorise the value, but the PE stills passes the information correctly to its neighbor.

ROL stands for "rotate left", ROU for "rotate up" and ROD for "rotate down". ROR is not defined.

ROW_ADR = ROW_IDX

Syntax: row_adr=row_idx

This instruction copies the ROW_IDX pointer to the current ROW_ADR. ROW_IDX is used as an original value of the row pointer, while ROW_ADR is the working value. There is no other effect.

ROW_IDX += NB_ROW

Syntax: row_idx+=nb_row

This instruction adds the contents of NB_ROW to ROW_IDX. ROW_IDX is used as an original value for the row pointer. However, to pass to the next tile, you must add NB_ROW to it.

SHIFT*xx*

Syntax: shift0
shift1
...
shift22
shift23
shift24

This controls the shifter that is between the memory and each FPGA. Each of the memory banks is 32bits in width, but you need only 8 to read at a given time. Shiftxx enables you to choose any of the 25 possible 8 bits out of 32.

STORE_COL

Syntax: store_col

The STORE_COL instruction instruct the FPGA #3 and #4 to write their outermost column (PE 48 to #63) to memory, in a location pointed by ADR. This instruction is used during preprocessing of the tiles.

MUTUALLY EXCLUSIVE INSTRUCTIONS

Each group represent instructions that can't be used simultaneously. Instructions that are in different groups can be used in the same cycle.

Group 1:	LOAD, ROL, ROU, ROD
Group 2:	CS0000, CS0001, ... , CS1111
Group 3:	STORECOL
Group 4:	LOADSIGMA
Group 5:	ENABLESIGMA
Group 6:	CLEARS, ENABLES, /* SAVES */
Group 7:	LOOP1, LOOP2, JUMP, CALL, RET, REP, EXIT
Group 8:	ADR=RESULT_ADR
	ADR=TILE_ADR
	ADR++
	ADR=COL_ADR
	ADR=TILE_ADR+IMAGE_WIDTH*2
	ADR=TILE_ADR+=2
	ADR=++COL_ADR
	ADR=++ROW_ADR
	ADR=--ROW_ADR
Group 9:	COL_ADR=COL_IDX
Group 10:	ROW_ADR=ROW_IDX
Group 11:	COL_IDX+=NB_COL
Group 12:	ROW_IDX+=NB_ROW
Group 13:	NB_REP=NB_COL-4
Group 14:	NB_LOOP1=IMAGE_WIDTH-2
Group 15:	NB_LOOP2=IMAGE_HEIGHT-2
Group 16:	SHIFT1, ..., SHIFT24
Group 17:	BREAKPOINT

Appendix 1: Error messages

Message	**Meaning**
Illegal mask length	The mask length is not 64.
Illegal character in mask	A character in the mask is not an hex digit.
Functional unit already allocated	You use an instruction that uses a functional unit that is already used by another instruction.
Syntax error	A word or token has not been recognised, such as a keyword that has been mistyped.
Includes cannot be nested	You can’t include a file within an include file.
Can’t open file *filename*	The file you have specified for including is either mispelled or inexistant.
label is used twice as a label	A label name must be unique.

end expected at line *line*	The parser finds a keyword that does not belong in the section being assembled, or end is mispelled.
Unresolved label *label* at line *line*	The label you referenced at line *line* can't be found in current source(s).
File not found or unreable	The file you have specified for assembly is either incorrectly specified or you may have unsufficient rights to access it (it might be protected against reading).

Références

[Ah86] Alfred V. Aho, Ravi Sethi et Jeffrey D. Ullman — Compilers: principles, techniques, and tools — Addisson-Wesley, 1986. (QA76.76 C65A37 1985).

[Al95] Altera — AHDL manual — Altera corporation, 1995, 2610 Orchard Park Way, San Jose.

[Ba92] Kurt Baudendistel — compiler development for fixed point processors — Ph.D. thesis, Georgia Institute of Technology, sept. 1992

[Cl94] Jocelyn Cloutier et Patrice Y. Simard — Hardware implementation of the backpropagation algorithm without multiplications — *in* Proceedings of the fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems, pp. 46-55, Torino, Italie, Sept. 1994.

[Cl95] Jocelyn Cloutier, Éric Cosatto, Steven Pigeon, François R. Boyer et Patrice Y. Simard — VIP: an FPGA-based processor for image processing and neural networks — *in* Proceedings of the fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems, pp.???-???, Lausane, Suisse, Fév. 1996.

[Co94] Éric Cosatto et Hans Peter Graf — NET32K: High speed Image Understanding System — *in* Proceedings of the fourth International Conference on Microelectronics for Neural Networks and Fuzzy Systems, pp. 413-412, Torino, Italie, Sept. 1994.

[G089] David E. Goldberg — Genetic algorithms in search, optimization and machine learning — Addisson-Wesley, 1992, 412p. (QA402.5 G635 1989)

[Ha94] Simon Haykin — Neural networks, a comprehensive foundation — MacMillan, 1994 (QA76.87 H39 1994)

[He91] John Hertz, Anders Krogh, Richard G. Palmer — Introduction to the theory of neural computation — *a lecture notes volume in the Santa Fe institute studies in the science of complexity*, Addisson-Wesley, 1991 (QA76.5 H475 1991)

[Hu92] Robert Hummel — Processeur et Coprocesseur: guide de référence du programmeur PC — Dunod, Paris, 1992.

[Is93] Jean-François Isabelle — Auto-apprentissage, à l'aide de réseaux de neurones, de fonctions heuristiques utilisées dans les jeux stratégiques — Mémoire, Université de Montréal, Décembre 1993, 50p.

[Ka93] Kai Hwang — Advaced computer architecture, parallelism, scalability, programmability — McGraw-Hill, 1993, 770p. (QA76.9 A73 H87 1993)

[Ki96] Kimihisa Aihara, Osamu Fujita et Kuniharu Uchimura — A digital neural network LSI using sparse memory architecture — *in* Proceedings of the fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems, pp.139-148, Lausane, Suisse, Fév. 1996.

[Ko90] Teuvo Kohonen — The self-organizing map — *Proceedings of the IEEE*, Vol 78, n9, Septembre 1990, pp. 1464-1480.

[Ku90] Ludek Kucera — Combinatorial algorithms — Adam Hilger, SNTL, Prague, 1990 (QA76.6 K8413 1989)

[Le90] Yan Le Cun, B. Boser, John S. Denker, D. Henderson, R.E. Howard, W. Hubbard et Larry D. Jackel — Handwritten digit recognition with a backpropagation network — Neural information processing system 2, 1990.

[Lu92] George F. Luger et William A. Stubblefield — Artificial intelligence, structures and strategy for complex problem solving — The Benjamin/Commings publishing Coe., 1992. 740p. (Q335 L84 1992)

[Ly94] Paul A. Lynn et Wolfgang Fuerst — Introductory digital signal processing with computer applications — John Wiley & Sons, 1994, 400p.

[Ly96] Richard F. Lyon et Larry S. Yaeger — On-line hand-printing recognition with neural networks — *in* Proceedings of the fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems, pp.201-212, Lausane, Suisse, Fév. 1996.

[Me92] Pankaj Mehra & Benjamin W. Wah, eds. — Artificial neural networks: concepts and theory — IEEE computer Society Press Tutorial, 1992 (QA76.87 A74 1992)

[Mi94] Philippe Michallon — Schémas de communications globales dans les réseaux de processeurs; applications à la grille torique — Thèse de doctorat, Institut national polytechnique de Grenoble, 1994

[Qu89] Patrice Quinton et Yves Robert — Algorithmes et architectures systoliques — Masson, Paris, 1989.

[Sä92] Edouard Säckinger, B.E. Boser, Jane Bromley, Yan LeCun et Larry D. Jackel — Application of the ANNA neural Network Chip to High-Speed Character Recognition — IEEE Transactions on Neural Networks, Vol. 3, no. 3, mai 1992.

[Si94] Patrice Y. Simard et Hans Peter Graf — Backpropagation without multiplication — *Neural Information Processing Systems*, Morgan Kaufmann, Vol 6, pp 232-239.