Rapport Projet DSP
Filtre de convolution

Bondu Martin 3700444
NEROT Stanislas 3711393

Professeur Référent: Habib Mehrez

1. Introduction

Depuis des années, le traitement d'images a été un des sujets porteur de l'informatique. Nécessitant des grandes quantités de données et de calculs, des méthodes de calcul sont vite apparues pour réaliser les différentes fonctions utilisées. Une des fonctions les plus connues est la convolution. Cette dernière est utilisée dans de multiples applications de filtres produisant des résultats divers.

Dans l'optique d'accélérer la réalisation de ces calculs (et bien d'autres du domaine plus large du traitement du signal) sont apparus des microprocesseurs spécialisés à cette fin : les DSP (Digital Signal Processors). En comparaison à du traitement de textes ou bases de données, le traitement du signal nécessite moins d'accès mémoire mais une plus grande parallélisation des calculs et données.

2. Objectifs du Projet

Le but du projet est d'implémenter une application de détection de contour sur un DSP: le T.I C6700. On cherche en réalité à mesurer l'intérêt d'une optimisation fine sur une portion de code critique, appelée de nombreuses fois : le calcul de convolution. On propose pour cela de s'intéresser à trois modèles d'implémentation :

- Code C
- Code Assembleur
- Code Assembleur Optimisé

et de comparer les performances de ces trois modèles à la simulation.

L'intérêt est donc double car il permet aussi d'explorer l'architecture de ce microcontrôleur, les techniques d'optimisation assembleur propre au C6700 et l'environnement de développement de ce composant.

3. Brève description de l'environnement de développement CCS

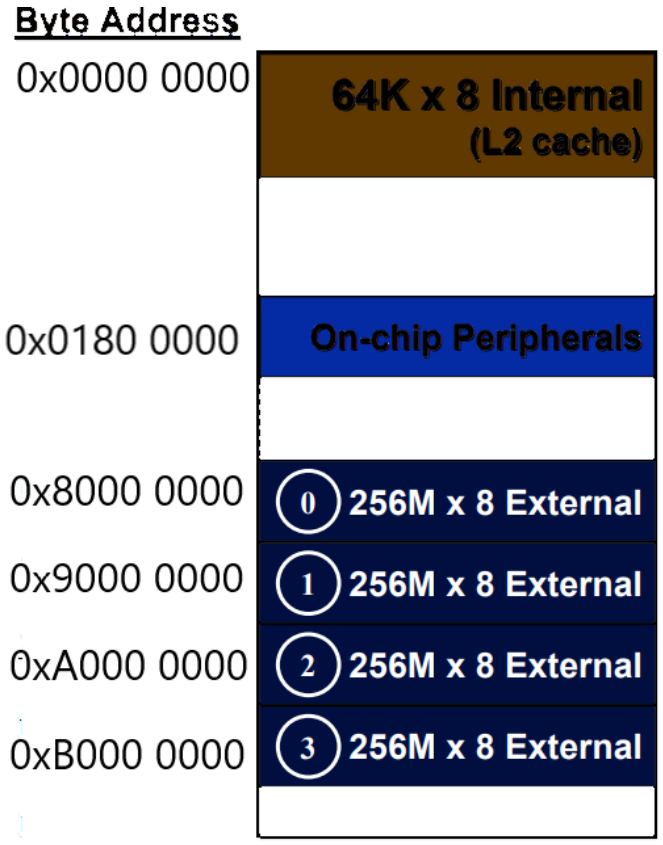
CCS est un environnement de développement pour les DSP dont le C6700 mais aussi les appareils basés sur l'architecture ARM. Il a été développé par Texas Instrument et basé sur le framework opensource Eclipse. Il fournit plusieurs outils pour faciliter la construction et la mise au point des programmes.

CCS consiste en un éditeur de code source en (C et assembleur), un compilateur C, un assembleur et un éditeur de liens, et un débogueur. Il fournit aussi un environnement de gestion de fichiers qui facilite la construction et la mise au point des programmes par un système de projet structuré par un fichier .pjt référençant les sources, librairies, éditeur de lien et en-têtes utilisées.
Il permet aussi l'analyse en temps réel de l'exécution des programmes de DSP par des mesures dans son outil de "Profiling".

4. *Brève description de l'architecture DSP utilisé*

Le C6700 est un processeur de traitement du signal multicore de la série TMS320 fabriqué par Texas Instruments. Il est principalement constitué de deux chœurs: A et B ayant chacun plusieurs unités de calculs.

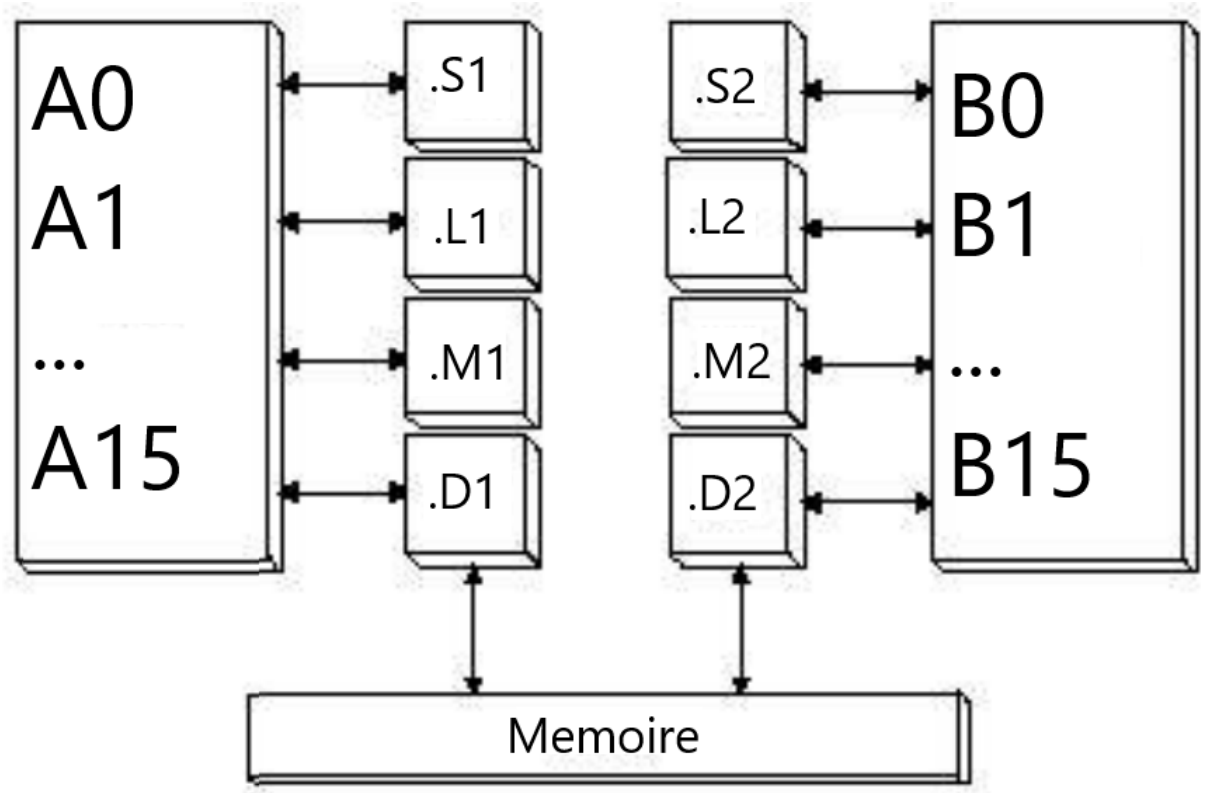
Il dispose d'un espace d'adressage de 2^{32} adresses et manipule des registres de 32 bits. La mémoire de ce microcontrôleur est organisé comme dessiné ci-dessous:



Chaque choeur dispose de 16 registres accessibles par le programmeur notés A_0 à A_{15} et B_0 à B_{15} . Les registres de 0 à 9 sont temporaires, 10 à 15 sont permanents (à l'exécution, on a expérimenté des comportements très étranges avec les registres permanents, qu'on a donc préféré éviter). Chaque choeur dispose de 4 unités de calcul réalisant différentes instructions :

- M : multiplications
- S : opérations logiques (OR, AND, XOR …), décalages (droite et gauche, signés ou non), saut, additions et soustractions
- L : opérations logiques (OR, AND, XOR …), comparaisons (>, <, =) , additions et soustractions
- D : Load, Store, additions et soustractions

Elles sont notées “.Ux” avec U l’initiale de l’unité, x=1 du côté A et x=2 du côté B, en voici une représentation :



Il peut donc réaliser jusqu’à 8 instructions de manière parallèle dont 2 multiplications et 2 opérations mémoire.

Il existe également des redirections pour utiliser un registre du banc A comme une opérande d’instruction effectuée sur une unité de B et l’inverse. Ces redirections sont appelées crosspath et on peut en utiliser au plus une en A et une en B simultanément par cycle. Les instructions peuvent utiliser une opérande supplémentaire appelée conditional et notée entre crochets avant l’instruction ex:

```
[conditional]  instruction opA opB opC
```

Ces conditions sont limitées aux registres A1, A2, B0, B1, B2. Si le registre mit en conditional est de valeur nulle, l’instruction ne s'exécute pas, sinon elle s'exécute.

5. *Développement*

Le tableau de pixels sera nommé source (composé d'unsigned char), et le filtre de convolution sera nommé pix_buf (composé de signed char).

a. *Format d'image traité*

Les fichiers PGM sont composés de :

- Un format
  - ☐ P2 pour un codage en ASCII
  - ☐ P5 pour un codage en binaire
- Une largeur
- Une hauteur
- Une valeur colormax
  - ☐ signifiant la valeur la plus haute de la grille de pixels afin d'améliorer l'affichage.
- Les valeurs de chaques pixels

Tous ces paramètres doivent être dans l'ordre ci-dessus et espacés entre eux d'un whitespace (espace, retour à la ligne, tabulation).
Des commentaires peuvent être mis n'importe où et doivent débuter avec #.

b. *Structuration du C*

Nous avons fait une boucle qui fait 9 itérations qui multiplie source[i] et pix_buf[i] et les additionne dans un accumulateur.

c. *Assembleur*

Similaire au code C, nous chargeons source[i] et pix_buf[i], puis les multiplions, les ajoutons à un accumulateur, puis décrémentons un compteur qui sert de condition sur un branchement permettant de reboucler.
Nous avons opté pour une implémentation simple, limitant le parallélisme, et donc respectant les temps d'exécution de chaque opération :

- 2 cycles pour une multiplication.
- 5 cycles pour une opération mémoire (load / store).
- 6 cycles pour un branchement.

d. *Assembleur optimisé (techniques d'optimisation)*

Plutôt que d'utiliser un pipeline software nous avons opté pour une implémentation de vectorisation de calculs nous permettant de mettre à profits les instructions LDW ainsi que d'utiliser les composants .M1 et .M2 en parallèle afin de faire plusieurs multiplications par cycles avec des masques sur les opérandes.

Nous utiliserons dans le reste de cette section les notations suivantes pour l'indice des pixels en fonction de leurs positions dans le filtre :

0	1	2
3	4	5
6	7	8

*le pixel 4 étant le pixel courant du programme principal.

Le principe général du programme est :

- Chargements
  - des pixels 0, 1, 2 et 3 de source et de pix_buf
    - des pixels 4, 5, 6 et 7 de source et de pix_buf
    - du pixel 8 de source et de pix_buf
    - des masques
- Traitements
  - On sépare les pixels par groupe de deux avec nos masques de façon à ce qu'il y ait un pixel du half fort et un pixel du half faible par groupe (on garde pour l'instant les séparations entre source et pix_buf)
    - pixels 0 & 3
    - pixels 1 & 2
    - pixels 4 & 7
    - pixels 5 & 6
  - On fait par groupe de deux pixels une multiplication sur le half faible (instruction MPY) et une multiplication sur le half fort (instruction MPYH) entre les pixels correspondant de source et de pix_buf
  - On shift les résultats qui sont décalés dûs au mauvais alignement et à l'utilisation des masques
  - On additionne tous les résultats (les 9 multiplications) entre eux dans A4
  - On branch sur l'adresse de B3

Explications techniques :

- Chaque chargement du tableau source par des LDW stocke les pixels comme suit :

Pixel 3	Pixel 2	Pixel 1	Pixel 0

  - après usage de nos masques, on obtient :
    - avec le masque 0xFF00 00FF

Pixel 3	0x00	0x00	Pixel 0

    - avec le masque 0x00FF FF00

0x00	Pixel 2	Pixel 1	0x00

  - ***Note: Pour notre algorithme il faut un pixel masqué dans le half fort et un dans le masque faible.***
  - ***Note 2: Nous avons choisis ces masques-ci car ils nous faisaient gagner un cycle à l'implémentation (permettant de faire des shifts plus tôt et donc de faire des additions plus tôt aussi).***
- Du côté du tableau pix_buf, nous ne pouvons pas faire le même traitement car c'est un tableau de ***signed*** *char*, nous avons donc opté pour une autre implémentation :
  - Nous utilisons l'instruction EXT sur chaque éléments du filtre pix_buf
    - l'instruction EXT provoque un shift left, suivi d'un shift right avec extension de signe.
      Il y a deux codages pour l'instruction EXT:
      - EXT (.unit) src2, csta, cstb, dst
        - les constantes ne peuvent pas être supérieur à 32, elles sont donc sur 5 bits
      - EXT (.unit) src2, src1, dst
        - src1 contient (csta<<5) + cstb
    - Exemple d'utilisation :
      - EXT .S2 B1, 24, 24, B1

0x00	0x00	0x00	0xFF

      - Résultat intermédiaire non visible

0xFF	0x00	0x00	0x00

      - Résultat obtenu

0xFF	0xFF	0xFF	0xFF

- Nous utilisons l'instruction EXT de sorte à toujours obtenir l'octet voulu avec signe étendu dans l'octet le plus à droite du registre.

- Nous multiplions les éléments correspondant de chaque tableau:
  - Exemple :

source[3]	0x00	0x00	source[0]

    - Si on multiplie le contenu de ce registre avec

sign extended	*sign extended*	*sign extended*	pix_buf[3]

    - on obtient avec MPYHL (half high du premier registre et half low du second)

0x00	*half fort*	*half faible*	0x00

  - Il nous faut donc shifter vers la droite les résultats des multiplications suivantes :
    - source[1] * pix_buf[1]
    - source[3] * pix_buf[3]
    - source[5] * pix_buf[5]
    - source[7] * pix_buf[7]
- On ajoute tous les résultats dans A4 (après les shift ci-dessus)
- On branche 5 cycles avant la fin afin d'éviter les NOP

On obtient 14 cycles en théorie

e. <u>Résumé des améliorations comparé au software pipelining</u>

En somme, les principes fondamentaux qui rendent notre solution plus performante que celle du software pipeline simple implémenté par nos camarades part d'un constat: la solution suggérée ne parallélise pas les multiplications.
On souhaite donc les paralléliser et les commencer au plus tôt .
Pour faire deux multiplications d'octets par cycle, il nous faut donc 4 octets disponibles par cycles et cela le plus tôt possible.
On fait donc deux LDW par cycle pour charger 16 octets par cycle qu'on masque ensuite pour avoir un seul octet par half dans les registres.
On arrive avec des shift avant ou après les multiplications à récupérer les bons résultats qu'on additionne en jonglant un peu avec les côtés A et B pour tout mettre dans A4 sans perdre de cycle à cause des dépendances de données.
La majorité des difficultées rencontrés sont liées aux extensions de signe, la limitation sur les cross paths et les shift ne se faisant qu'en unité S.
On note qu'avec l'instruction MPYU4 présente sur les C6000 mais pas les C6700, on aurait pu gagner 5 cycles en économisant 5 MPY et un cycle d'additions.

6. *Mise en oeuvre*

a. *Résultats C pur (profiling)*

Ayant effectué la mesure sur une image de *168* pixels, nous avons mesuré *648760* cycle total ce qui nous donne donc :
648760 / *168* = *3861* cycles /pixels

b. *Résultats C avec assembleur (profiling)*

Ayant effectué la mesure sur une image de *168* pixels, nous avons mesuré *161634* cycle total ce qui nous donne donc :
161634 / *168* = *962* cycles /pixels

c. *Résultats C avec assembleur optimisé (profiling)*

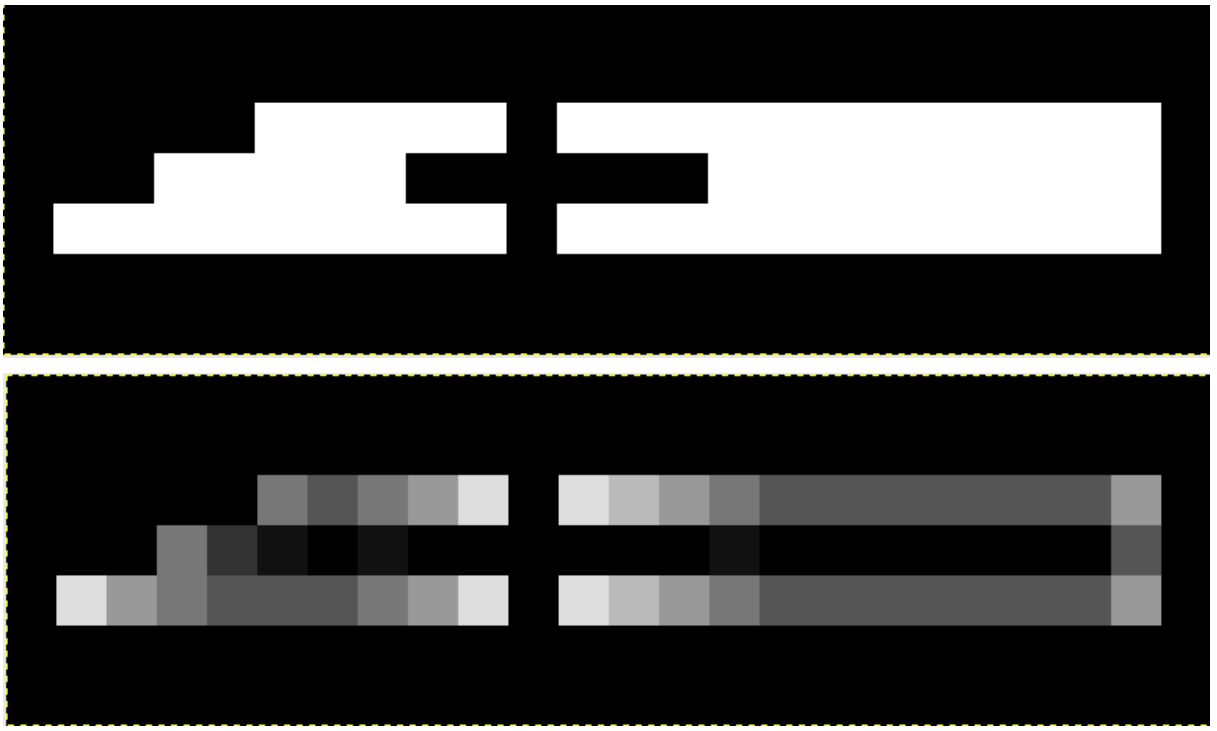
Ayant effectué la mesure sur une image de *168* pixels, nous avons mesuré *3523* cycle total ce qui nous donne donc :
3523 / *168* = *20* cycles /pixels

d. *Exemples de résultats images*

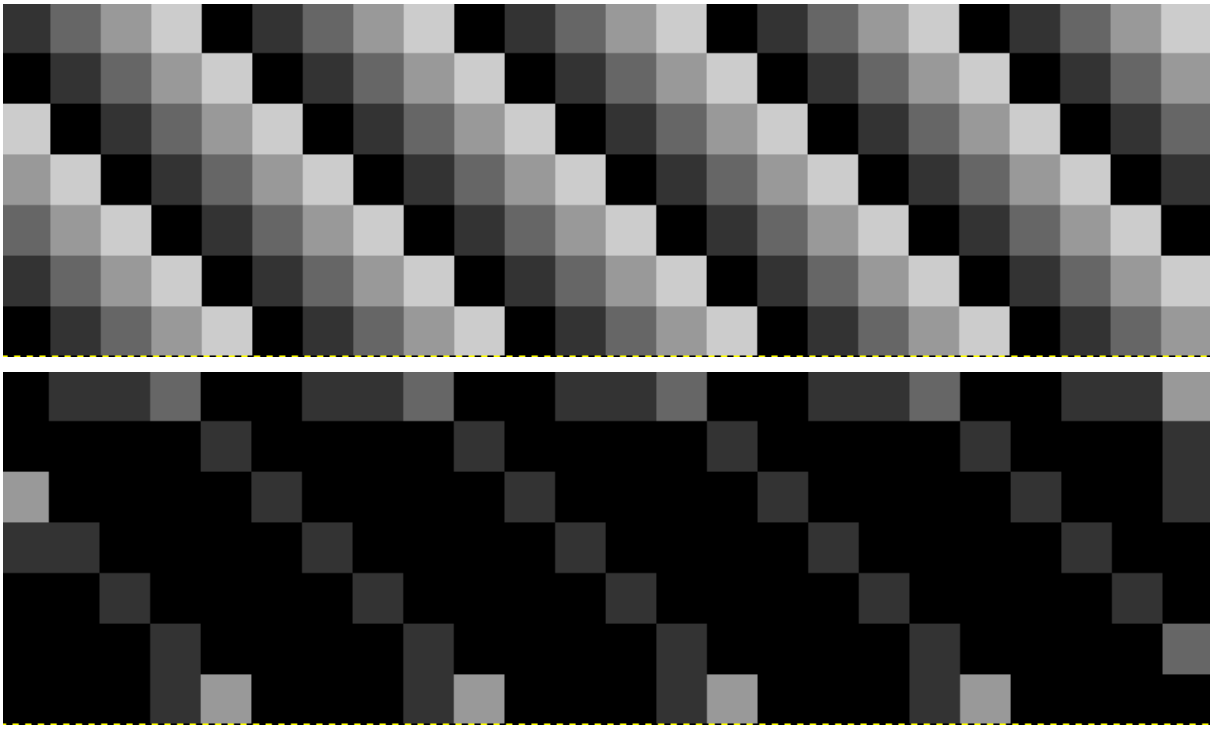
Les trois fonctions donnent évidemment un résultat identique (testé sur plusieurs images), nous avons donc décidé de ne faire qu'une seule section pour les trois codes.

Images initiale simple avec résultats en dessous:
Carré blanc sur fond noir:

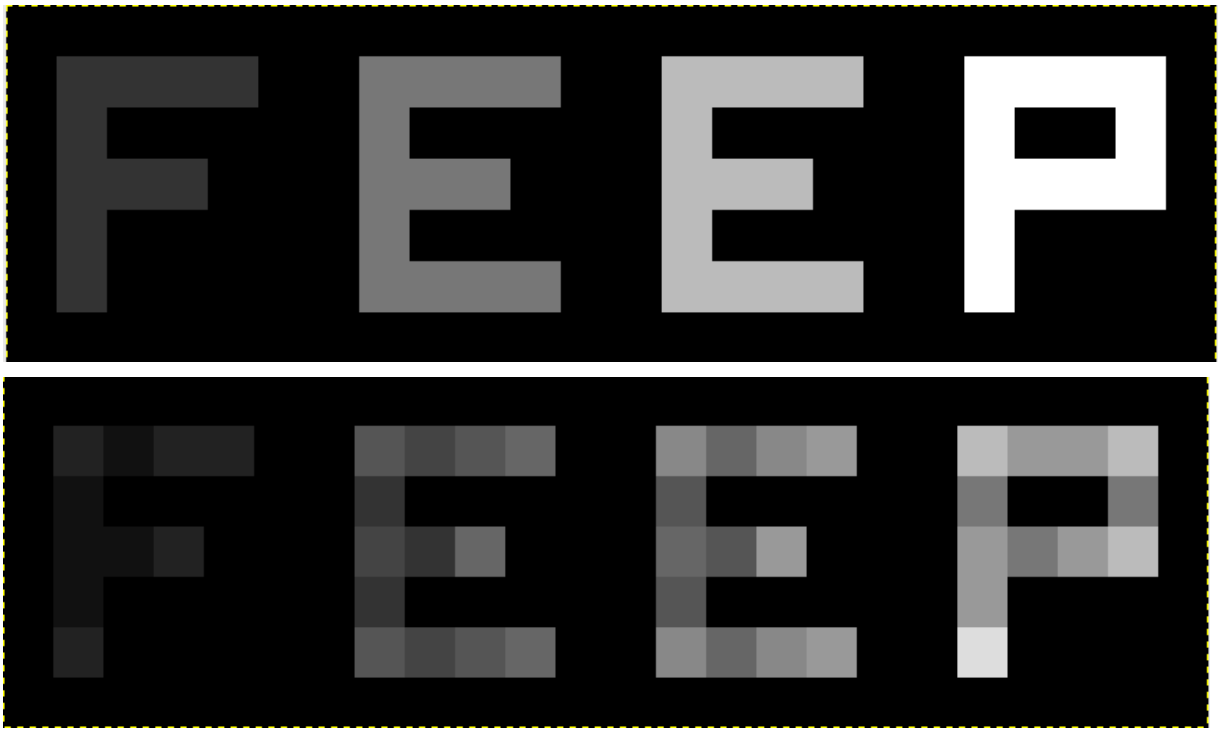
Croix noir sur escalier blanc, dans une salle mal éclairée :



Écran de télévision cathodique dysfonctionnel :



Logo FEEP page principale de Netpbm:



e. *Comparaison des performances des 3 précédentes méthodes*

	Code C	**Assembleur**
Assembleur	4x plus rapide	
Assembleur Optimisé	193x plus rapide	48x plus rapide

7. Conclusion

On peut dégager deux conclusions assez évidente de ces résultats :

Tout d'abord le fait qu'adapter un code à une architecture DSP en le passant en assembleur rend le traitement plus efficace car il permet de passer par les registres pour les variables intermédiaires réduisant les accès mémoire et accélèrant grandement l'application (faisable aussi avec une optimisation du C avec des `volatile` .. un meilleur compilateur est aussi une option).

D'autre part l'intérêt flagrant d'optimiser cet assembleur puisqu'on arrive, sur un petit code à diviser par 50 le temps d'exécution par rapport à l'assembleur non opti. Ce bout de code étant appelé de très nombreuses fois, on économise ici sur une très petite image une centaine de milliers de cycles.

Note: Malgré de nombreuses difficultées rencontrées pour trouver une optimisation meilleure que celle du software pipeline, le chemin parcouru nous a permis de comprendre en profondeur certaines spécificités non documentées du C6700. Nous espérons que ce code puisse servir aux générations suivantes d'élèves en DSP pour leur montrer une alternative plus performante de ce calcul de convolution et nous serions honorés qu'il puisse être évoqué ou même présenté dans votre cours :-).