# Transformations algorithmiques

Lionel Lacassagne

LIP6

Sorbonne University (UPMC) / LIP6 / ALSOC

https://www.lip6.fr/

lionel.lacassagne@lip6.fr

# Comment aller plus vite ?

- Optimiser la durée des boucles
  - car c'est là que se concentre la majorité des calculs
  - $\Rightarrow$ identifier les boucles consommant le plus de temps
  - alors comment aller plus vite ?
- Réduire la durée des calculs
  - en enchainant mieux les calculs (archi & pipeline)
  - en diminuant leur nombre (algo & factorisation)
  - en les remplaçant par des calculs équivalents plus rapides (algo & maths)
- Réduire la durée des accès mémoire
  - en enchainant mieux les accès mémoire (archi & mémoire cache)
  - en diminuant leur nombre (algo)
  - c'est aujourd'hui *la* priorité
- Les architectures actuelles
  - ont une énorme puissance de calcul (SIMD $\times$ multicoeurs)
  - qui n'est utilisable que si les données sont proches des unités de calcul

# *Memory layout*

- Deux principaux type de stockage des données: AoS et SoA
- Exemple: ensemble de coordonnées 3D $(x, y, z)$
- AoS = *Array of struct*
  - approche C et C++: 1 tableau contenant $n$ structures
  - stockage mémoire: $x_0, y_0, z_0, x_1, y_1, z_1, x_2, y_2, z_2, \cdots, x_{n-1}, y_{n-1}, z_{n-1}$
  - accès `T[i].x`, `T[i].y`, `T[i].z`
- SoA = *Struct of Array*
  - approche Fortran: autant de tableau que d'éléments dans la structure
  - stockage mémoire: $X = [x_0, x_1, x_2, \cdots, x_{n-1}]$ $Y = [y_0, y_1, y_2, \cdots, y_{n-1}]$, et $Z = [z_0, z_1, z_2, \cdots, z_{n-1}]$
  - accès `X[i]`, `Y[i]`, `Z[i]`
- AoSoA = *Array of Struct of Array* ou Hybrid-SoA
  - approche SIMD: SoA par paquet (équivalent au cardinal du registre SIMD)
  - stockage mémoire: $T = [x_0, x_1, x_2, x_3, y_0, y_1, y_2, y_3, z_0, z_1, z_2, z_3, x_4, x_5, x_6, x_7, \cdots]$

# Fusion d'opérateurs

## réduire le nombre d'accès mémoire

- Fusion de boucles

```
for(i=0;i<n;i++)
  Y[i] = f(X[i]);
for(i=0;i<n;i++)
  Z[i] = g(Y[i]);

for(i=0;i<n;i++) {
  Y[i] = f(X[i]);
  Z[i] = g(Y[i]);
}
```

- Fusion d'opérateurs

```
for(i=0;i<n;i++) {
  Z[i] = g(f(X[i]);
}
```

- Simple si
  - composition de fonction au sens mathématique: $y = f(x), z = g(y), z = (g \circ f)(x)$
  - tant que les fonctions / opérateurs sont ponctuels
  - ex: $f(x) = 2x, g(x) = x + 1, (g \circ f)(x) = 2x + 1$

# Fusion d'opérateurs

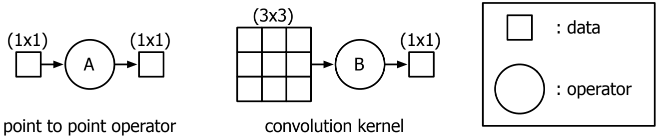
## modèle producteur-consommateur

- Complexe
  - lorsque les opérateurs ont un voisinage (*stencils*, convolutions)
- Exemple (encore simple): stencil 1D
  - $F : y(n) = x(n) + x(n-1)$
  - on veut connaitre $H = G \circ F$ (pour simplifier, $G = F$)
  - on a donc: $z(n) = y(n) + y(n-1)$
  - en développant: $z(n) = [x(n) + x(n-1)] + [x(n-1) + x(n-2)]$
  - soit $z(n) = x(n) + 2x(n-1) + x(n-2)$ c'est le triangle de Pascal
- Modèle producteur-consommateur
  - Design Pattern célèbre en système (concurrence)
- Exemple 0: fonctions ponctuelles
  - si $y = f(x) = 2x$, on a: input = 1 point, output = 1 point, on note $1 \rightarrow 1$
  - si $z = g(y) = y + 1$, on a: input = 1 point, output = 1 point, on note $1 \rightarrow 1$
- Règle de composition
  - lorsque les modèles producteurs-consommateurs sont compatibles
  - lorsque la sortie du premier opérateur à la même forme que l'entrée du second (trivial pour les fonctions)

# Fusion d'opérateurs

## modèle producteur-consommateur

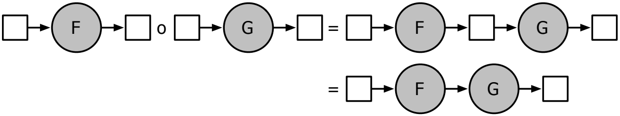
- Exemple 1: stencils 1D
  - on a $F : y(n) = x(n) + x(n-1)$ soit $2 \to 1$
  - on a $G : z(n) = y(n) + y(n-1)$ soit $2 \to 1$
  - il faut donc que $F$ soit appelée 2 fois pour produire 2 fois 1 point,
  - pour que $G$ soit ensuite appelée
- Prologue et épilogue
  - problème traitée dans la partie sur le pipeline d'opérateurs
- Fusion d'opérateurs 2D
  - on considère 2 type d'opérateurs
  - opérateur ponctuel $1 \to 1$ soit $(1 \times 1) \to (1 \times 1)$
  - opérateur de convolution ou stencil $(k \times k) \to (1 \times 1)$



# Fusion d'opérateurs

## règles de composition

- Cas #1: composition d'opérateurs ponctuels
  - $(1 \times 1) \rightarrow (1 \times 1) \circ (1 \times 1) \rightarrow (1 \times 1)$

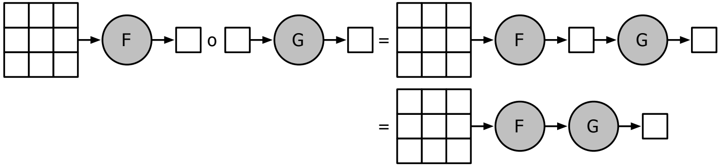



- Rien à faire de particulier, on obtient
  - $(1 \times 1) \rightarrow (1 \times 1)$
  - 1 appel de $F$ et 1 appel de $G$

# Fusion d'opérateurs

## règles de composition

- Cas #2: composition d'un opérateur ponctuel et d'un stencil
  - $(3 \times 3) \rightarrow (1 \times 1) \circ (1 \times 1) \rightarrow (1 \times 1)$

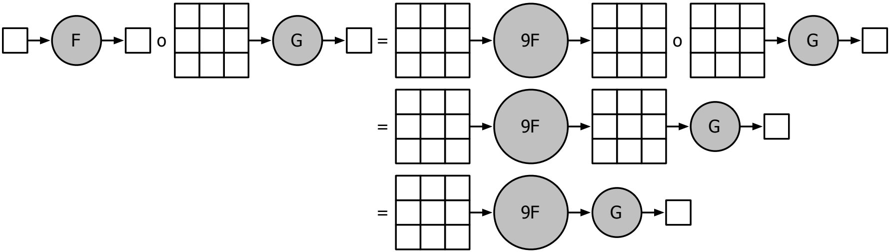



- Rien à faire de particulier, on obtient:
  - $(3 \times 3) \rightarrow (1 \times 1)$
  - 1 appel de $F$ et 1 appel de $G$

# Fusion d'opérateurs

## règles de composition

- Cas #3: composition d'un stencil et d'un opérateur ponctuel
  - $(1 \times 1) \rightarrow (1 \times 1) \circ (3 \times 3) \rightarrow (1 \times 1)$

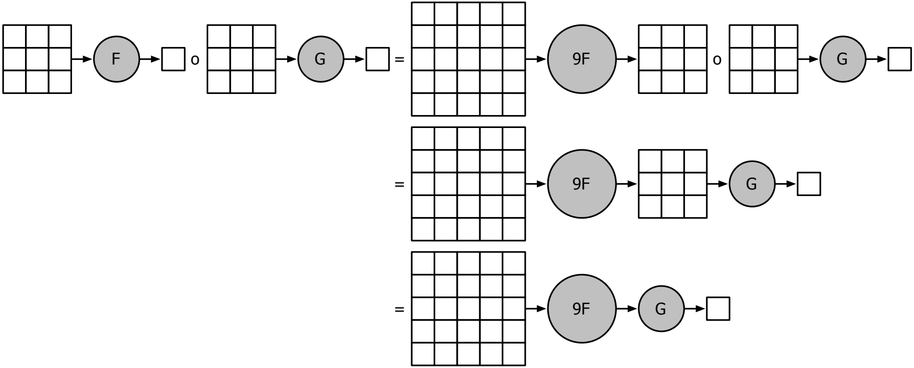



- Adaptation nécessaire des patterns d'entrée / sortie
  - adapter le pattern de sortie de $F$ pour correspondre au pattern d'entrée de $G$
  - $F$ doit être appelée $(3 \times 3)$ fois pour produire suffisamment de données
  - même pattern que pour le cas #2 mais avec 9 appels de $F$ et 1 appel de $G$

# Fusion d'opérateurs

## règles de composition

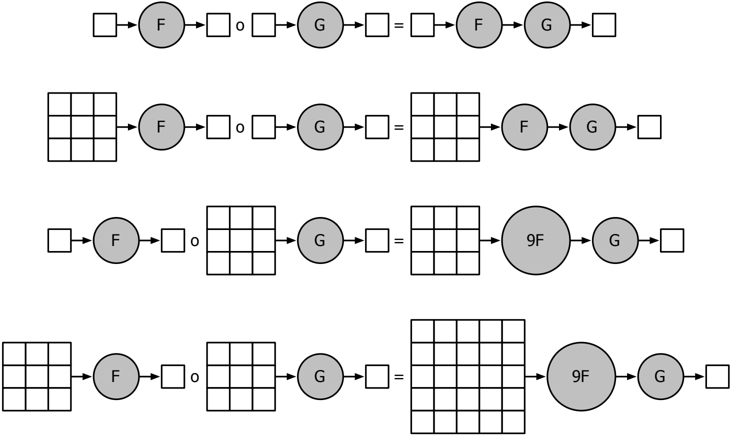
- Cas #4: composition de 2 stencils
  - $(3 \times 3) \rightarrow (1 \times 1) \circ (3 \times 3) \rightarrow (1 \times 1)$




- Adaptation nécessaire des patterns d'entrée / sortie
  - adapter le pattern de sortie de $F$ pour correspondre au pattern d'entrée de $G$
  - comme pour le cas #3, il y a 9 appels de $F$ et 1 appel de $G$
  - mais le pattern d'entrée change: $(5 \times 5) \rightarrow (1 \times 1)$

# Fusion d'opérateurs

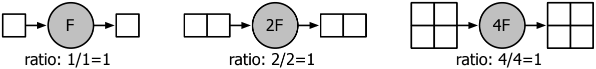
## règles de composition



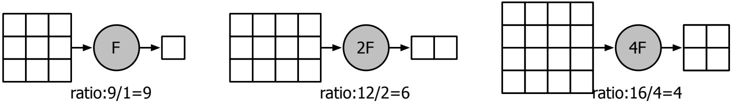
# Duplication d'opérateurs

## déroulage de boucle

- Nouvelle métrique d'estimation de performance
  - *cpp* (cycle par point) $\rightarrow$ *rpp* (read per point)
  - nombre de lectures (de read ou de load) pour produire 1 point
  - par extension, ratio du nombre de lectures par le nombre d'écritures
- Déroulage de boucle + ré-use de data
  - lorsqu'on déplace le stencil sur les données, il y a recouvrement et ré-use
  - les points déjà chargés peuvent être utilisés une seconde fois
    il suffit de dérouler la boucle




- Opérateur ponctuel: pas de gain




- Stencil/convolution: le gain augmente avec le déroulage
  - la complexité aussi ...

# Opérateurs séparables

## + rotation de registres / déroulage de boucle

- Un grand nombre d'opérateurs de convolution et de stencils sont séparables

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 \end{bmatrix}$$

- Code naïf

```
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    Y[i] = X[i-1][j-1] + X[i-1][j] +
           X[i  ][j-1] + X[i  ][j];
```

- Code naïf scalarisé

```
for(i=0;i<n;i++)
  for(j=0;j<n;j++) {
    a0=X[i-1][j-1]; b0=X[i-1][j];
    a1=X[i  ][j-1]; b1=X[i  ][j];
    ra=a0+a1; rb=b0+b1; // operateur colonne
    y=ra+rb;            // operateur ligne
    Y[i][j]=y;
  }
```

# Opérateurs séparables #1

## deux balayages

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 \end{bmatrix}$$

- Deux balayages avec tableau temporaire `T`
  - stencil vertical parcours vertical + stencil horizontal parcours horizontal

```
for(j=0;j<n;j++)
  for(i=0;i<n;i++)
    T[i][j] = X[i-1][j] + X[i][j];  // parcours vertical
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    Y[i][j] = T[i][j-1] + T[i][j];  // parcours horizontal
```

- Deux balayages avec tableau temporaire `T`
  - stencil vertical parcours horizontal + stencil horizontal parcours horizontal

```
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    T[i][j] = X[i-1][j] + X[i][j];  // parcours horizontal
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    Y[i][j] = T[i][j-1] + T[i][j];  // parcours horizontal
```

# Opérateurs séparables #2

## fusion des deux balayages en un seul

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 \end{bmatrix}$$

- Deux balayages avec tableau temporaire `T`

```
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    T[i][j] = X[i-1][j] + X[i][j];  // parcours horizontal
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    Y[i][j] = T[i][j-1] + T[i][j];  // parcours horizontal
```

- Un balayage avec tableau temporaire `T`

```
for(i=0;i<n;i++)
  for(j=0;j<n;j++) {
    T[i][j] = X[i-1][j] + X[i][j];
    Y[i][j] = T[i][j-1] + T[i][j];
  }
```

# Opérateurs séparables

## + rotation de registres / déroulage de boucle

- Code scalarisé avec rotation de registres naïve

```
for(i=0;i<n;i++) {
  j=0;
  a0=X[i-1][j-1];
  a1=X[i  ][j-1];
  for(j=0;j<n;j++) {
    b0=X[i-1][j];
    b1=X[i  ][j];
    ra=a0+a1; rb=b0+b1; // operateur colonne
    y=ra+rb;            // operateur ligne
    Y[i][j]=y;
    a0=b0; a1=b1;       // *2* RR
  }
}
```

# Opérateurs séparables

## + **réduction** par colonne

- Code scalarisé avec rotation de registres prenant en compte le ré-use
  - réduction par colonne = application de l'opérateur 1D vertical

```
for(i=0;i<n;i++)
  j=0;
  a0=X[i-1][j-1];
  a1=X[i  ][j-1];
  ra=a0+a1;            // reduction de la premiere colonne
  for(j=0;j<n;j++) {
    b0=X[i-1][j];
    b1=X[i  ][j];
    rb=b0+b1;          // reduction de la nouvelle colonne
    y=ra+rb;           // operateur ligne
    Y[i][j]=y;
    ra=rb;             // *1* RR
  }
}
```

- Transformations
  - recopie de l'indice de boucle à l'extérieur de la boucle
  - déplacement des chargements et des calculs des premières colonnes de l'intérieur à l'extérieur de la boucle

# Opérateurs séparables

## + **réduction** par colonne + déroulage

- Code scalarisé avec réduction et déroulage

```
for(i=0;i<n;i++)
  j=0;
  a0=X[i-1][j-1];
  a1=X[i  ][j-1];
  ra=a0+a1;                 // reduction de la premiere colonne

  for(j=0;j<n-r;j+=2) {  // epilogue manquant

    b0=X[i-1][j];
    b1=X[i  ][j];
    rb=b0+b1;               // reduction de la nouvelle colonne
    y0=ra+rb;               // operateur ligne
    Y[i][j+0]=y0;

    a0=X[i-1][j+1];
    a1=X[i  ][j+1];
    ra=a0+a1;              // reduction de la premiere colonne
    y1=rb+ra;              // operateur ligne
    Y[i][j+1]=y1;
  }
}
```

- impossible à écrire sans scalarisation

# Opérateurs séparables

## Analyse de la complexité

| version | ADD | LOAD+STORE | IA | ratio |
|---|---|---|---|---|
| naïve | 3 | $4+1=5$ | 0.6 | 4:1 |
| rotation | 3 | $2+1=3$ | 1.0 | 2:1 |
| réduction | 2 | $2+1=3$ | 0.6 | 2:1 |
| réduction + déroulage | $2\times 2=4$ | $2(2+1)=6$ | 0.6 | $4:2=2:1$ |

- Rotation
  - la rotation seule ne fait – en général – pas gagner beaucoup de temps
- Réduction
  - permet de faire baisser à la fois la complexité et le nombre d'accès mémoire
  - pas un problème si l'IA reste basse, car le nombre total d'accès mémoire a diminué
  - l'impact de la réduction augmente avec la taille des opérateurs
- Réduction + déroulage
  - supprime la rotation des variables réduites
  - peut limiter l'impact de la latence des opérations et donc accélérer la boucle

# Pipeline d'opérateurs ponctuels

réduire la durée des accès mémoire en maximisant le ré-use du cache

- Parfois, il est impossible de faire de la fusion d'opérateurs
  - typiquement: deux filtres récursifs parcourant les données en sens opposé
- On réalise alors un pipeline d'opérateurs
  - c'est certes moins efficace qu'une fusion
  - mais cela améliore la persistance des données en cache
  - cela reste donc très performant
- Fusion de boucles (des boucles externes)
  - on suppose que les boucles internes ne sont pas fusionnables
  - ou que leur fusion est inefficace voire contre-productive
  - à chaque fois qu'une ligne est produite, elle est aussitôt re-consommée
  - exemple de pipeline d'opérateurs ponctuels (simple)

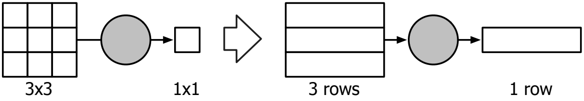
```
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    Y[i][j]=f(X[i][j]);
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    Z[i][j]=g(Y[i][j]);

for(i=0;i<n;i++) {
  for(j=0;j<n;j++) Y[i][j]=f(X[i][j]);
  for(j=0;j<n;j++) Z[i][j]=g(Y[i][j]);
}
```

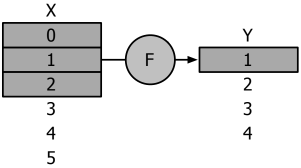
# Pipeline de stencils/convolution

## exemple simple

- Pipeline de 2 stencils $F$ et $G$ de pattern $(3 \times 3) \rightarrow (1 \times 1)$



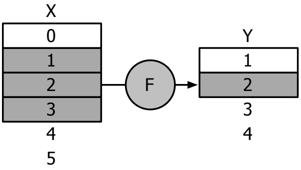

- Problème
  - le voisinage va nécessiter un découpage spécial de la boucle
  - avec un prologue et (parfois) un épilogue
- Prologue



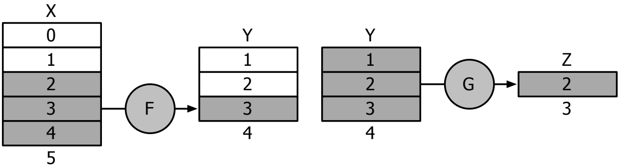
# Pipeline de stencils/convolution

## exemple simple

- Prologue



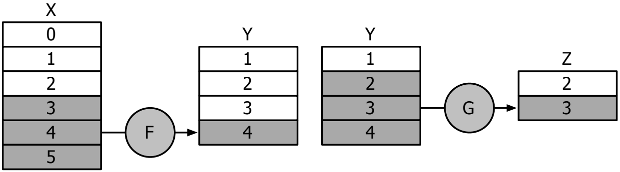

- Début de la boucle



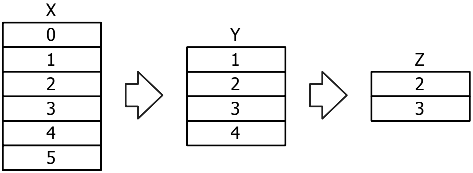
# Pipeline de stencils/convolution

## exemple simple

- Suite de la boucle



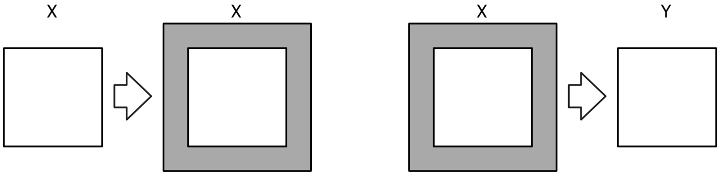

- En résumé
  - Pour un stencil de diamètre $k \times k$ (avec $k = 2r + 1$ et $r$ le rayon)
  - A chaque matrice/image perd $2r$ lignes par rapport à la précédente
  - Il est nécessaire d'avoir une/des stratégies pour gérer les bords haut et bas (aka bords horizontaux) ...
  - ... en plus des bords gauche et droit (aka bords verticaux)



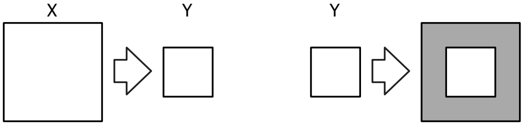
# Gestion des bords pour un stencil

## cas d'un stencil unique

- Plusieurs stratégies possibles
  - ajouter des bords pour que le code reste simple (un nid de boucle unique)
  - ne pas ajouter des bords (car cela n'est pas possible) et écrire un code complexe (erreur prone) : le nid de boucle + le traitement des 4 bords + le traitement des 4 coins = 9 instances du corps de boucle ...
- pré-duplication des bords



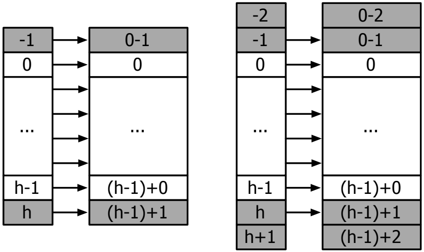

- post-duplication des bords



# Gestion des bords pour un stencil: bords haut et bas

## cas d'un stencil unique

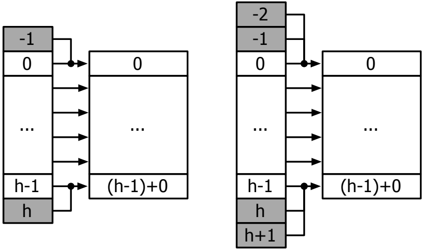
- 2 cas à distinguer
  - la duplication des bords verticaux qui peut être évitée via RR ou LU
  - la duplication des bords horizontaux qui ne peut être évitée via RR ou LU
- matrice sans bord vertical mais avec bords horizontaux
  - gauche: stencil de diamètre 3 (et de rayon 1) `matrix(0-1,h-1+1,0,w-1);`
  - droite: stencil de diamètre 5 (et de rayon 2) `matrix(0-2,h-1+2,0,w-1);`



# Gestion des bords pour un stencil: bords virtuels

## cas d'un stencil unique

- Matrice sans bord vertical et avec bords horizontaux virtuels
  - gauche: stencil de diamètre 3 (et de rayon 1) `matrix_vbord(0,h-1,0,w-1, 1);`
  - droite: stencil de diamètre 5 (et de rayon 2) `matrix_vbord(0,h-1,0,w-1, 2);`




- Des avantages
  - plus d'allocation supplémentaire
  - plus de duplication
  - fonctionne aussi avec des données déjà allouées: il suffit de faire un nouveau wrapper de lignes
- un inconvénient
  - nécessite toujours RR ou LU pour chaque ligne

# Gestion des bords pour une composition de stencils

- CEAN
  - C/C++ Extension for Array Notation
  - Intel `https://software.intel.com/en-us/cpp-compiler-developer-guide-and-r`
  - Cilk: `https://www.cilkplus.org/tutorial-array-notation`

- Version simplifiée pour expliquer un traitement et faire disparaitre un niveau de boucle
  - ligne $i$: $X(i) \equiv X_i \equiv$ `X[i]`
  - traitement d'une ligne: $Y_i = f(X_i) \equiv$ `Y[i]=f(X[i])`
  - `Y[i]=f(X[i]) for(j=0;j<w;j++) Y[i][j]=f(X[i][j]);`
  - sans se préoccuper du traitement de la ligne (bord, RR, LU, ...) car problème orthogonal

- Version simple: pré-duplication des bords et traitement standard

```
X[-1]=X[0]             // bord haut
X[(h-1)+1]=X[(h-1)]    // bord bas
for(i=0;i<h;i++) {     // i decrit l'espace d'arrivee
  Y[i]=f(X[i]);        // traitement de la ligne i
}
```

# Gestion des bords pour un pipeline de stencils

- Pseudo-code sans pipeline.
  - avec potentiellement un cache-miss car le bord bas est initialisé avant le parcours de X

```
X[-2]=X[-1]=X[0]                  // bord haut
X[(h-1)+2]=X[(h-1)+1]=X[(h-1)]    // bord bas
for(i=0-1;i<=(h-1)+1;i++) {       // Y[0-1:(h-1)+1]
  Y[i]=f(X[i-1],X[i],X[i+1])      // X[0-2:(h-1)+2]
}
for(i=0;i<=(h-1);i++) {           // Z[0:(h-1)]
  Z[i]=g(Y[i-1],Y[i],Y[i+1])      // Y[0-1:(h-1)+1]
}
```

- Pseudo-code sans pipeline
  - avec parcours totalement dans l'ordre

```
X[-2]=X[-1]=X[0]                    // bord haut
for(i=0-1;i<=(h-1);i++) {           // Y[0:h-1]
  Y[i]=f(X[i-1],X[i],X[i+1])        // X[0-1:h-1+1]
}
X[(h-1)+2]=X[(h-1)+1]=X[(h-1)]                        // bord bas
Y[(h-1)+1]=g(X[(h-1)+0],X[(h-1)+1],X[(h-1)+2])        // y[(h-1)+1]
for(i=0;i<=(h-1);i++) {                               // Z[0:h-1]
  Z[i]=g(Y[i-1],Y[i],Y[i+1])                          // Y[0-1:h-1+1]
}
```

# Gestion des bords pour un pipeline de stencils

- Pseudo-code avec pipeline (pour matrices avec bords)

```
X[-2]=X[-1]=X[0] // bord haut

// prologue pipeline
Y[-1]=f(X[-2],X[-1],X[0])
Y[ 0]=f(X[-1],X[ 0],X[1])

// pipeline
for(i=0;i<=(h-1)-2;i++) {           // Z[0   :h-1-2]
  Y[i+1]=f(X[i  ],X[i+1],X[i+2])   // Y[0+1:h-1-2+1] Y[0+1:h-1-1]
  Z[i  ]=g(Y[i-1],Y[i  ],Y[i+1])   // X[0+0:h-1-2+2] X[0+0:h-1-1]
}

X[h-1+2]=X[h-1+1]=X[h-1+0] // bord bas

// epilogue pipeline
Y[h-1+0]=f(X[h-1-1],X[h-1+0],X[h-1+1])  // avant dernier points
Z[h-1-1]=g(Y[h-1-2],Y[h-1-1],Y[h-1+0])

Y[h-1+1]=f(X[h-1+0],X[h-1+1],X[h-1+2]) // dernier points
Z[h-1  ]=g(Y[h-1-1],Y[h-1+0],Y[h-1+1])
```

# Gestion des bords pour un pipeline de stencils

- Pseudo-code avec pipeline (pour matrices avec bords virtuels)