Problèmes d'utilisation des instructions SIMD

- Les problèmes liés à la nature des instructions SIMD
  - Formats d'entrée doivent être homogènes
  - Format de sortie doit être le même que le format d'entrée
    - Problème intrinsèque avec les formats entiers
  - Les formats d'entrées doivent être adaptés au parallélisme de données
- Les insuffisances dans le jeu d'instructions SIMD Intel *SSE 2*
  - Problèmes d'alignement mémoire
  - Manque de certaines instructions (addition horizontale « réduction »)

Problèmes de l'arithmétique entière

- Problème spécifique aux instructions SIMD entières
  - Addition : N bits + N bits donnent N+1 bits
    - soit arithmétique saturée
    - soit complément à 2 avec PERTE DE LA RETENUE
  - Multiplication : N * N donnent 2N bits
- Instructions spéciales de multiplication ou multiplication - accumulation
  - Multiplication N*N et résultat sur 2N bits (avec éventuellement accumulation)
    - SSE :
      - PMADDWD : 16 X 16 +32
      - PMULUDQ : 32 x 32 => 64
    - Altivec:
      - plusieurs variantes 16 x 16 +32
      - plusieurs variantes de multiplications 8 x 8 => 16

Valeur absolue

- Il n'y a pas d'instruction SIMD de valeur absolue
  - Calcul de la valeur absolue du contenu d'un registre SIMD
    - `__m128i v1;`
    - Créer une constante SIMD zéro
    - Calculer zéro –v1
    - Calculer max (v1, zéro – v1)

Vectorisation automatique par le compilateur

- Le compilateur va chercher à vectoriser le code avec l'option :
  - Compilateur Intel : `-xSSE2` ou `-xAVX` …
  - Gcc : `-msse2` ou `-mavx` ...
- Les obstacles intrinsèques à la vectorisation :
  - Problèmes fondamentaux de vectorisation/parallélisation

    Ex : les dépendances de données entre itérations
- Le compilateur va vectoriser une boucle (la plus interne) si il peut s'assurer qu'il peut traiter les instructions par bloc (dans des vecteurs).
- Il faut donc éviter :
  - les types : pointeurs ou structures
  - les accès aux tableaux avec pas non unitaires…
  - les tests *(if)*
  - que le nombre d'itérations ne soit pas connu lors de l'exécution
  - …

Problèmes intrinsèques - exemple : Deriche

- Deriche horizontal (flottants)

```
for(i=0; i<size; i++) {
   for(j=0; j<size; j++) {
      Y[i][j] = b0 * X[i][j] + a1 * Y[i][j-1]
         + a2 * Y[i][j-2] ;}
   for (j=size-1;j>=0;j--) {
      Y[i][j] = b0 * X[i][j] + a1 * Y[i][j+1]
         + a2 * Y[i][j+2]; } }
```

Y(i, j) = f [(Y(i,j-1), Y(i,j-2)]

Dépendance entre itérations

- Deriche vertical (flottants)

```
for(j=0; j<size; j++) {
   for(i=0; i<size; i++) {
      Y[i][j] = b0 * X[i][j] + a1 * Y[i-1][j]
         + a2 * Y[i-2][j];}
   for (i=size-1;i>=0;i--) {
      Y[i][j] = b0 * X[i][j] + a1 * Y[i+1][j]
         + a2 * Y[i+2][j];} }
```

Pas non unitaire pour
la boucle interne

Vectorisation automatique

- Conditions de « vectorisation »
  - Accès à pas unitaire
  - Pas de dépendances de données
  - Pas de pointeurs
- Performance des caches
  - Accès à pas unitaire

Éviter les pointeurs

- Pointeurs et incrémentation/décrémentation de pointeurs n'est pas autorisé pour indexer les boucle

```
int a[100];
int *p;
p=a;
for (i=0; i<100;i++)
      *p++ = i;
```

Ne vectorise pas

```
int a[100] ;


for (i=0; i<100;i++)
      a[i] = i;
```

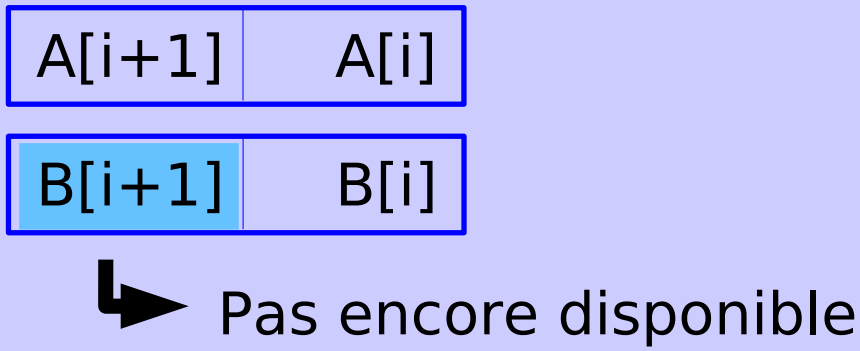
Vectorise

Dépendances propagées entre itérations

S1: A[i] = A[i] + B[i];

S2: B[i+1] = C[i] + D[i];

Pas de vectorisation



S2: B[i+1] = C[i] + D[i]

S1: A[i+1] = A[i+1] + B[i+1]

Vectorisation

Échange de boucles

```
void loop_interchange_example(float *a, float *b, float *c)
{
    for (int j=0; j<100; j++)
        for (int i=0; i<100; i++)                NV
            a[i,j]=a[i,j] + b[i,j] * c[i,j]      pas ≠ 1
}
void loop_interchange_example(float *a, float *b, float *c)
{
    for (int i=0; i<100; i++)
        for (int j=0; j<100; j++)                V
            a[i,j]=a[i,j] + b[i,j] * c[i,j]      pas = 1
}
```

Éclatement de boucles

```
void spliting_example(float *a, float *b, float *c, float *d, float *e, float *e,
   float *g)
    {    for (int i=0; i<99; i++)
         {   a[i]= c[i] + e[i] * b[i];                    NV
             b[i]= x + a[i] + g[i];
             c[i+1] = a[i] + b[i] + f[i];
         }
    }

         for (int i=0; i<99; i++)
             d[i]= e[i] * b[i];                          Vectorisable
         for (int i=0; i<99; i++)
         {   a[i]= c[i] + d[i];                          NV
             b[i]= x + a[i] + g[i];
             c[i+1] = a[i] + b[i] + f[i];
         }
    }
```

Fusion de boucles

```
void loop_fusion_example(float *a, float *b, float *c, float *d)
    {
        for (int i=0; i<100; i++)
            a[i]= b[i] + c[i];                          V
        for (int i=0; i<100; i++)
            d[i]= a[i] * 2;                             V
    }

    for (int i=0; i<100; i++)
    {
        a[i]= b[i] + c[i];                              V
        d[i]= a[i] * 2;                                 réduit le surcoût
    }
```

Copie de variable

```
for (int i=0; i<100; i++)
{    a[i]= (b[i] + b[i+1])/2;
     b[i+1] = c[i];                    NV
}
```

```
for (int i=0; i<100; i++)
     d[i]= b[i+1];                     V
for (int i=0; i<100; i++)
{    a[i]= (b[i] + d[i])/2;            NV
     b[i+1] = c[i];
}
```

```
for (int i=0; i<100; i++)
     d[i]= b[i+1];
for (int i=0; i<100; i++)
{    b[i+1] = c[i];
     a[i]= (b[i] + d[i])/2;
}
```

Application : filtre

- Caractéristiques
  - stockage des images (octets)
  - traitement (short ou int)
  - résultat (octets)
- exemples : filtre, gradient, morpho-math...
  - Filtre de Deriche :

```
unsigned char X[size][size], Y[size][size];
int b0, a1, a2;
for(i=0; i<size; i++) {
    for(j=0; j<size; j++) {
        Y[i][j] =(unsigned char) (b0 * X[i][j] + a1 * Y[i][j-1] + a2 * Y[i][j-2]) >> 8);}
    for (j=size-1;j>=0;j--) {
        Y[i][j] = (unsigned char) (b0 * X[i][j] + a1 * Y[i][j+1] + a2 * Y[i][j+2]) >> 8); } }
```

Application : filtre

- Solution Deriche Horizontal-Vertical (flottants) :

```
for(i=0; i<size; i++) {
   for(j=0; j<size; j++) {
      Y[i][j] = b0 * X[i][j] + a1 * Y[i-1][j] + a2 * Y[i-2][j];}}
for (i=size-1;i>=0;i--) {
   for(j=0; j<size; j++) {
      Y[i][j] = b0 * X[i][j] + a1 * Y[i+1][j] + a2 * Y[i+2][j];}}
```

  - pas de dépendances entre itérations
  - pas unitaire

  → Vectorisation automatique en flottants

Application : filtre

- Deriche entiers : formats différents

```
byte **X, **Y;
int32 b0, a1, a2;
for(i=0; i<size; i++) {
   for(j=0; j<size; j++) {
      Y[i][j] = (byte) (b0 * X[i][j] + a1 * Y[i-1][j] + a2 * Y[i-2][j]) >> 8);}}

for (i=size-1;i>=0;i--) {
   for(j=0; j<size; j++) {
      Y[i][j] = (byte) (b0 * X[i][j] + a1 * Y[i+1][j] + a2 * Y[i+2][j]) >> 8);}}
```

Pas de dépendance
Pas unitaire
mais 8 bits et 32 bits

- Deux problèmes
  - formats différents
  - multiplications SIMD : 16x16 → 32 bits

→ Vectorisation automatique impossible
Intrinsics ou assembleur éventuellement

Entiers ou Flottants ?

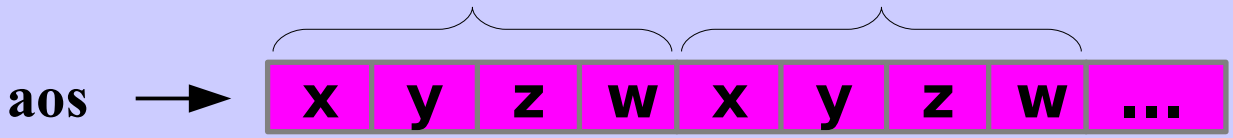
- Entiers :
  - Pb de calculs intermédiaires
  - Changements de formats 8 – 16 – 32 bits
    - Formats identiques
      - Connaissance de l'amplitude des valeurs
      - Perte de précision
    - Formats différents
      - Pas de vectorisation automatique
      - Vectorisation « éventuelle » à la main
- Flottants
  - Coût de mémorisation des pixels 8bits → flottants (8 → 32bits)
  - Conversion entre formats entiers-flottants

Entiers ou Flottants ?

- Entiers : Pb de précision
  - Par exemple : x = y * 1/3 + z * 2/5
    - Si y et z valent 1 le résultat sera 0
    - Solution : x = (y * 5 + z * 6)/15
  - Comment augmenter la précision du calcul sur entiers ?
    - Changer de formats 8 → 16 , 16 → 32 bits
      - Augmente l'amplitude des valeurs
    - Utiliser la virgule fixe
      - Décalage à gauche de 8 ou 16 bits de toutes les opérandes
        → y* 256 ou y*65536
      - Calcule sur les opérandes
      - Décalage à droite de 8 ou 16 bits du résultat
        → x/ 256 ou x/65536

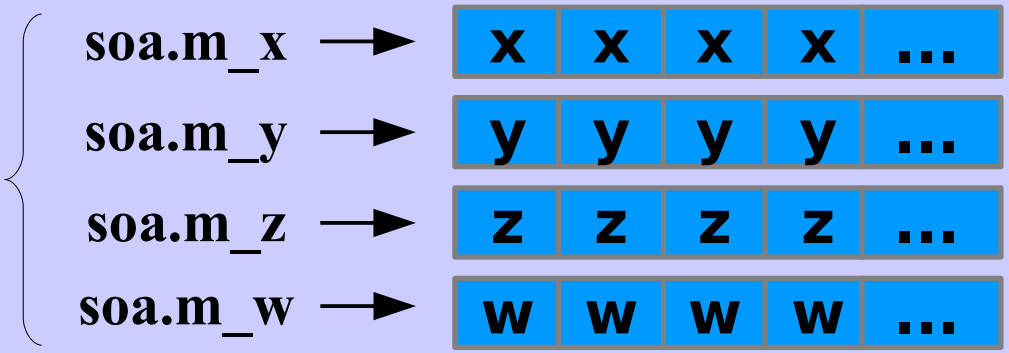
SIMD : structuration des données

- AoS : *Array of Structures*



pas adapté aux SIMD

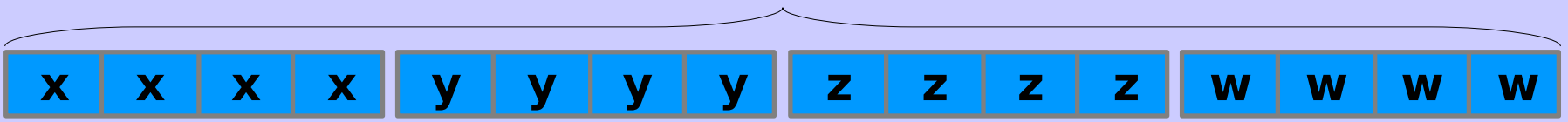
- *SoA : Structure of Arrays*



permet de paralléliser

- *HSoA : Structure Hybride*

accès mémoire plus efficace



SIMD : structuration des données

- AoS : *Array of Structures*

```
struct {
   float x, y, z, r, g, b;
} AoS_xyz_rgb[200];
```

- *SoA : Structure of Arrays*

```
struct {
   float x[200], y[200], z[200];
   float r[200], g[200], b[200];
} SoA_xyz_rgb;
```

- *HSoA : Structure Hybride*

```
struct {
   float xx[4], yy[4], zz[4];
} Hybrid_xyz[50];
struct {
   float rr[4], gg[4], bb[4]
} Hybrid_rgb[50];
```

SIMD ARM : NEON

ARM : Advanced Risc Machines Ltd.

- Fondé en Novembre 1990 (Acorn Computers) UK
- Conception des processeurs RISC ARM
- Vente de licences de conception des processeurs ARM aux fabricants qui fondent et vendent les puces.
- ARM ne fabrique rien
- Développe des technologies pour aider à la conception d'architecture ARM
  - Outils logiciel, conseil, matériel de mise au point, applications, périphériques, etc

SIMD ARM : NEON

- Famille de processeurs ARM Cortex, partageant le même *core* 32 bits et le même jeu d'instruction de base ARMv7
- Cortex-A8 : haute performance, faible consommation :

  → jeux, communication sans fil, routers, multimédia embarqué
- Intègre NEON
  - Co-Processeur dédié multimédia (MM)
  - Architecture SIMD 64/128 bits
    - 32 registres 64 bits ou 16 registres 128 bits
    - données sur 8/16/32/64bits
    - support optimale pour pixels, nombre complexes, etc.
    - mémoire considérée comme tableau de structures (AoS)

SIMD ARM : NEON

- NEON : *pipeline* de 10 étages
- Les adresses des données MM manipulées par les *load*/*store* sont calculées par le *pipeline* entier
- Une file d'instructions « *Neon Instruction Queue NIQ* » pour découplé ARM/NEON. Chacun avance à sa vitesse.
- La plupart des instructions NEON s'exécutent en 1 cycle
- Exécution en parallèle (même cycle) des instr de calcul et de transfert (permutation ou accès mémoire).
- Différent du jeu d'instr. SIMD ARM v6 32bits, moins avancé

SIMD ARM : NEON

- Types de Données :
  - 32-bit single precision floating-point
  - 8, 16, 32 and 64-bit unsigned and signed integers
  - 8 and 16-bit polynomials *(CRC, crypto...)*

unsigned integer	U8 U16 U32 U64
signed integer	S8 S16 S32 S64
integer of unspecified type	I8 I16 I32 I64
floating-point number	F16 F32 (or F)
polynomial over {0,1}	P8 P16

SIMD ARM : NEON

- Registres :
  - D0-D31 : 32 registres 64 bits
  - Q0-Q15 : 16 registres 128 bits
  - D0-D1 <=> Q0
- Opérandes des instructions :
  - *Normal* : opérandes en entrées et en sortie de même type/taille
  - *Long* : sortie de même type mais du double de la taille des entrées Q*d* ← D*n*, D*m*
  - *Wide* : 1ère opérande en entrée et sortie double de la taille de la 2nd opérande en entrée Q*d* ← Q*n*, D*m*
  - Narrow: sortie de même type mais de moitié de la taille des entrées D*d* ← Q*n*, Q*m*

SIMD ARM : NEON

- Accès aux données 8, 16 ou 32 bits entrelacées en mémoire

  Instructions *Vector Load/Store* selon 4 modes :

  - VLD1 : accès aux données contiguës – normal
  - VLD2 : données chargées une sur deux dans deux registres différents

    par exemple pour séparer les deux voix d'un son stéréo
  - VLD3 : données désentrelacées dans trois registres différents

    par exemple pour séparer les couleurs de pixels dans une image : RGB
  - VLD4 : données désentrelacées dans quatre registres différents

    par exemple pour séparer les couleurs de pixels dans une image : RGBA

SIMD ARM : NEON - ACLE

- ACLE : ARM C Language Extensions
  - Fonctions *Intrinsics* C/C++ pour – entre autre – NEON

```
#ifdef __ARM_NEON
#include <arm_neon.h>
#endif /* __ARM_NEON */
```

```
int32x4_t       // vecteur de 4 entiers de 32 bits
uint16x8_t      // vecteur de 8 entiers naturels de 16 bits
int8x8x2_t      // banque de 2 vecteurs de 8 entiers de 8 bits
```

  - Option de compilation gcc : -mfpu=neon

DLU: *Deep Learning Unit*

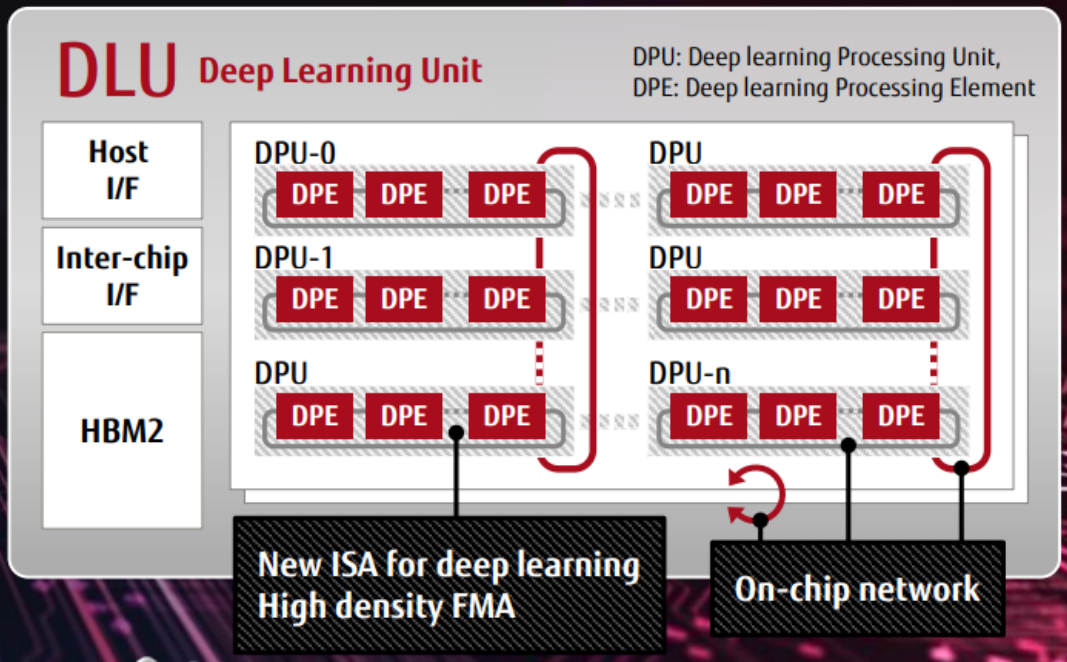
- Fujitsu : processeur dédié à l'apprentissage automatique
  - optimise la gestion des données en FP32, FP16, INT16 et INT8. Fujitsu a démontré que ce type de calculs offrait d'excellents résultats, malgré une précision moins importante
  - rapport performances/Watt dix fois supérieur à la concurrence
  - architecture autour de cœurs DPU (Deep Learning Processing Unit).
  - chaque DPU intègre 16 DPE (Deep Learning Processing Element)
  - chaque DPE offre huit unités d'exécution SIMD, et un grand banc de registres au lieu d'une mémoire cache L1 traditionnelle
  - 1 mémoire HBM2 partagée comme cache centralisé
  - Sortie en 2018. coprocesseur pour CPU SPARC déjà installés dans les supercalculateurs de Fujitsu

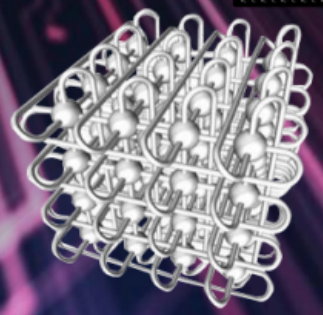
DLU: *Deep Learning Unit*

DLU Architecture

FUJITSU

- ISA: Newly developed for deep learning
- Micro-Architecture
  - Simple pipeline to remove HW complexity
  - On-chip network to share data between DPUs
- Utilizes Fujitsu's HPC experience, such as high density FMAs and high speed interconnect
- Maximizes performance / watt





Fujitsu's interconnect technology

Large scale DLU interconnect through off-chip network