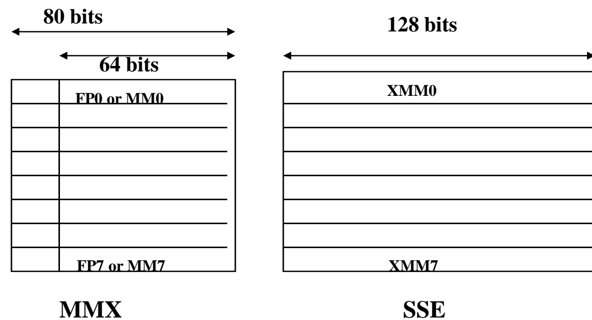
Extensions SIMD dans les microprocesseurs

Daniel Etiemble
de@lri.fr

Extensions SIMD dans les jeux d’instructions

- Dans tous les jeux d’instructions
  - MIPS, PowerPC AltiVec, ARM (Neon)
  - IA-32 (Intel MMX, SSE, SSE2,SSE3, SSE4), Intel 64 (AVX) et AMD
- Applications
  - Audio, Communication, Noyaux DSP, Graphique 2D et 3D, Images, Vidéo, Reconnaissance parole, etc
- Formes limitées d’instructions vectorielles
  - Vecteurs courts (2, 4, 8, 16 éléments) sur 128 bits (ou 256 bits)
  - Accès mémoire avec pas unitaire
    - Pas de scatter-gather, pas d’accès avec pas non unitaire
  - Transfert registres, registres mémoire et opérations sont SIMD (tous les éléments du vecteur simultanément)
  - Instructions “spéciales” multimédia
    - Ex : Somme de valeurs absolues de différences

Les registres SIMD (IA-32)

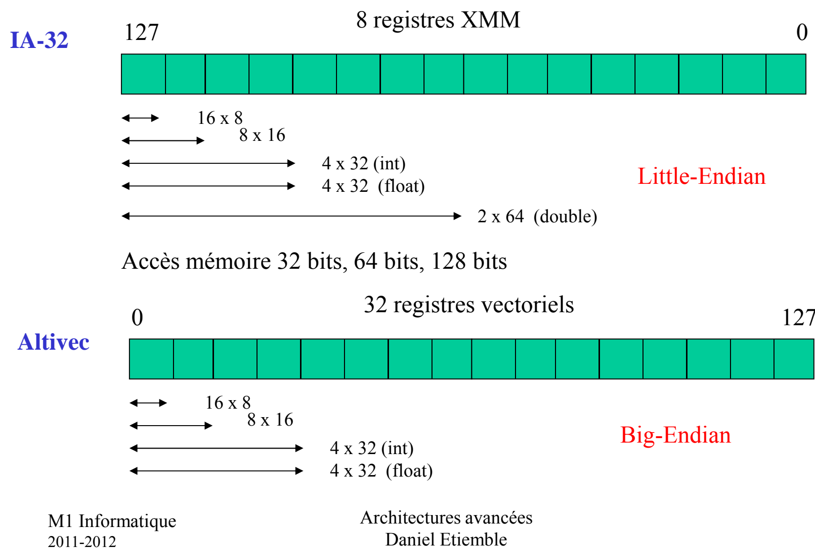


- Les registres MMX sont partagés avec les registres flottants (pile x87)
- Les registres XMM sont distincts

Les registres SIMD (Intel 64)

- Intel AVX
  - 16 registres 256 bits Ymm0 à Ymm15
  - Formats
    - SSE = AVX 128 bits
    - 8 floats
    - 4 doubles
  - Implanté dans « Sandy Bridge » (32 nm)

Formats de données SIMD



Utilisation des instructions SIMD

- C (ou Fortran)
  - Transformation du code pour rendre les boucles vectorisables
- Intrinsics
  - Appel de fonctions de type C
  - Le compilateur traite l'allocation des registres et l'ordonnancement
- Langage assembleur
  - Assembleur avec instructions SIMD dans le code C

Exemples d'intrinsics

```
__m128 _mm_set_ps1(float f)

__m128 _mm_load_ps(float *mem)

__m128 _mm_mul_ps(__m128 x, __m128 y)

__m128 _mm_add_ps (__m128 x, __m128 y)

void _mm_store_ps(float *mem, __m128 x)
```

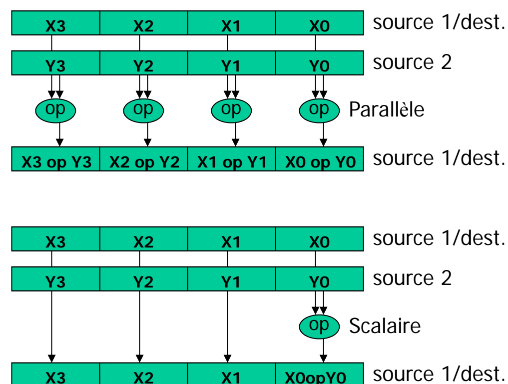
Exemple : l’inversion d’images

```
void inversion(byte **X, long i0, long i1, long j0, long j1, byte **Y)
{  int i, j;  for(i=i0; i<=i1; i++)
              {    for(j=j0; j<=j1; j++)
              {      Y[i][j] = 255 - X[i][j]; }  }}
```

```
void inversionS(byte **X, long i0, long i1, long j0, long j1, byte **Y)
{  int i, j, m, n;
  __m128i **XS, **YS;
  __m128i constante, l1, l2;
 XS=X; YS=Y;
constante =_mm_set1_epi8 (-1);
for(i=i0; i<=i1; i++) {
   for(j=j0; j<=j1/16-1; j++) {
              _mm_store_si128 (&YS[i][j], _mm_subs_epu8 (constante,XS[i][j] ));}}}
```


Instructions SIMD IA-32 parallèles et scalaires

- Instructions arithmétiques et logiques
- Instructions mémoire
- Instructions de formatage et manipulation
- Instructions de conversion



SIMD IA-32 : opérations arithmétiques

Arithmétique entière	Complément à 2	Saturation signée	Saturation non signée
Addition	PADD(b,w,d)	PADDS(b,w)	PADDUS(b,w)
Soustraction	PSUB(b,w,d)	PSUBS(b,w)	PSUBUS(b,w)
Multiplication	PMUL(llw,lhw)		
Multiplication accumulation	PMADDWD		

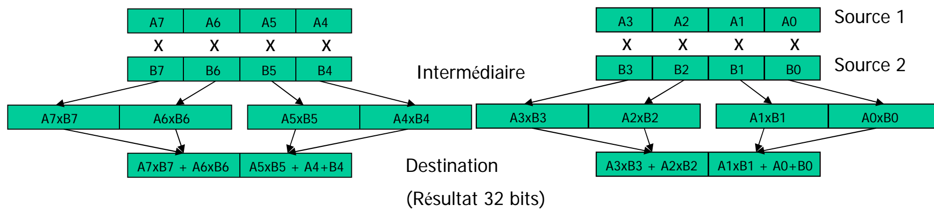
Arithmétique flottante	Parallèle SP	Scalaire SP	Parallèle DP	Scalaire DP
Addition	ADDPS	ADDSS	ADDPD	ADDSD
Soustraction	SUBPS	SUBSS	SUBPD	SUBSD
Multiplication	MULPS	MULSS	MULPD	MULSD
Division	DIVPS	DIVSS	DIVPD	DIVSD
Racine carrée	SQRTPS	SQRTSS	SQRTPD	SQRTSD

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 et résultat sur 2N bits (avec éventuellement accumulation)
    - IA-32
      - 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

Multiplication-accumulation entière

- 8 multiplications et 4 additions en une instruction PMADDWD

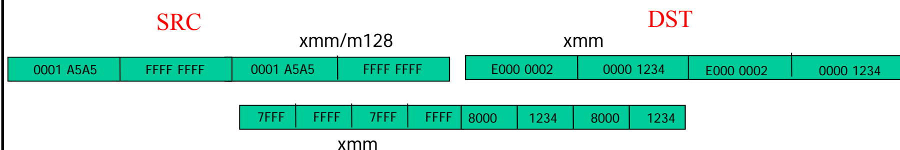


- PMADDWD produit 2 résultats 32 bits
  - Utile pour les applications multimédia et traitement du signal
  - Formats d'entrée et de sortie différents
  - N'existe pas pour les données 8 bits en entrée

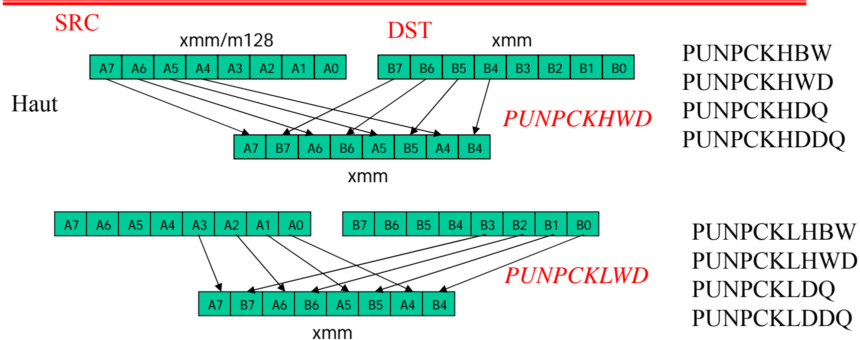
Compactage-décompactage

Arithmétique entière	Compl. à 2	Saturation signée	Saturation non signée
Pack		PACKSS(wb,dw)	PACKUS(wb)
Unpack High	PUNPCKH(bw,wd,dq)		
Unpack Low	PUNPCKL(bw,wd,dq)		

- PACKSSDW – Compacte les données 32 bits en données 16 bits avec saturation signée (plus grand/plus petit si débordement par défaut ou par excès). Utile pour les données 16 bits avec calcul sur précision 32 bits.

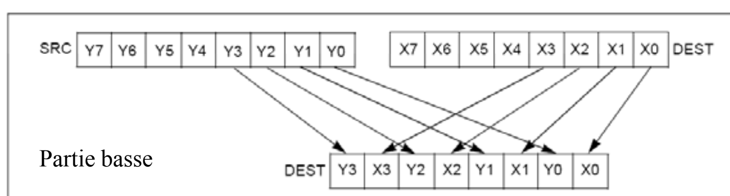


Décompactage



- Utile pour convertir des données, rassembler des données, entrelacer/dupliquer des données, transposer des lignes et des colonnes

Passage de 8 bits à 16 bits

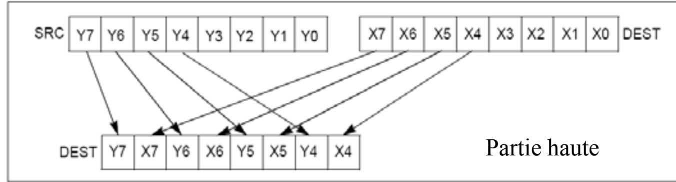


SOURCE : 00 00 00 00 00 00 00 00

DESTINATION : 12 34 56 78 9a bc de f0

DESTINATION : 009a 00bc 00de 00f0

Passage de 8 bits à 16 bits



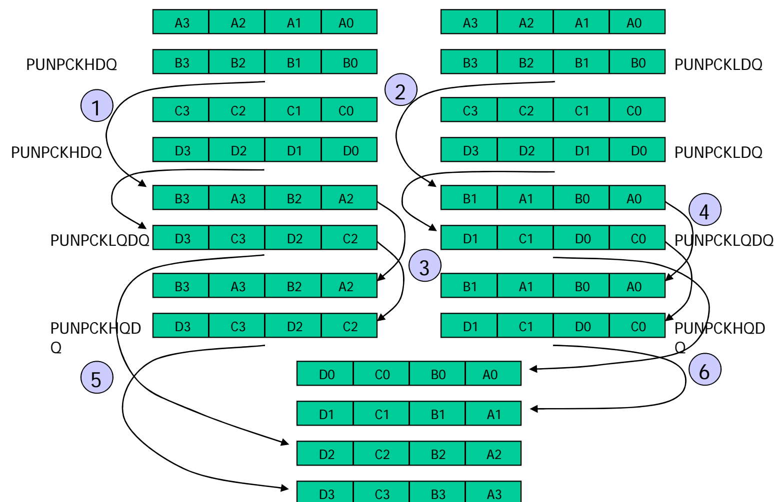
Partie haute

SOURCE : 00 00 00 00 00 00 00 00

DESTINATION : 12 34 56 78 9a bc de f0

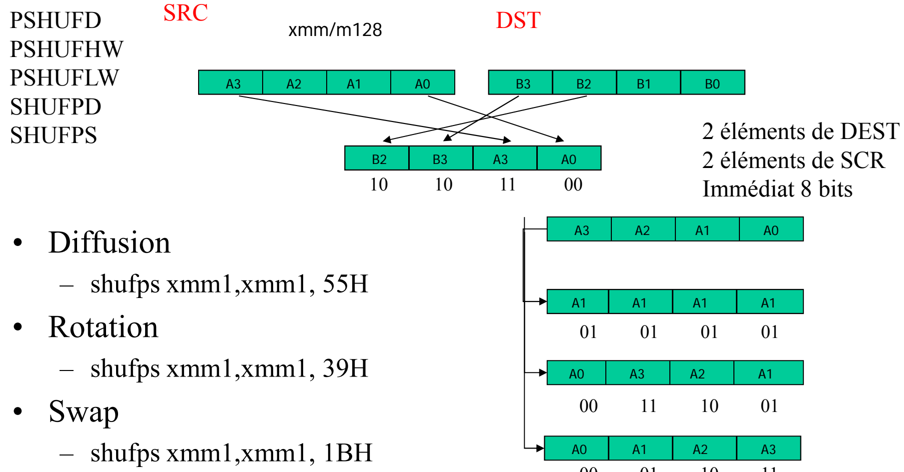
DESTINATION : 0012 0034 0056 0078

Transposition de matrice 4x4 avec Unpack



Les instructions “shuffle”

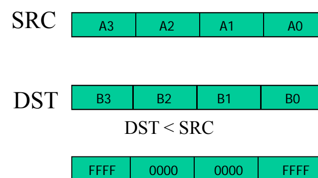
PSHUFD
PSHUFHW
PSHUFLW
SHUFPD
SHUFPS



- Diffusion
  - shufps xmm1,xmm1, 55H
- Rotation
  - shufps xmm1,xmm1, 39H
- Swap
  - shufps xmm1,xmm1, 1BH

Comparaisons et opérations logiques

- Compare (eq, lt, le, unord, neq, nlt, nle, ord) and set mask
  - Flottants
    - cmp*xx*ps ou cmp*xx*ss
    - cmp*xx*pd ou cmp*xx*sd
  - Entiers
    - PCOMPEQ (B,W,D)
    - PCOMPGT (B,W,D)
- Opérateurs logiques
  - and, andn, or, xor
    - versions ps et ss
    - Version entière



Conditionnelles SIMD

Exemple

```
unsigned char X[512], max;
        max = X[0];
   for (i=1 ; i<512 ; i++)
        if (X[i] >max) max=X[i] ;
```

Programme Scalaire

Suppression des branchements conditionnels

Version SIMD

```
unsigned char X[512], max, byte[16];
_m128i XS[32], M , a;
XS=X ;
M= ld16(&XS[0]) ;
  for(i=0; i<32; i++) {
            a = ld16(&XS[i]);
            M = maxbu(M,a);_}

byte=&M ;
max = byte[0]
for (i=1 ; i<16 ;i++)
            if (byte[i] >max) max=byte[i];
```


Valeurs absolues

- Si x(i) <0 alors |x(i)| = -x(i)
- Il n’y a pas d’instruction SIMD de calcul de la valeur absolue
- Calcul des valeurs absolues du contenu d’un registre SIMD
  - __m128i v1;
  - Créer une constante SIMD zero
  - Calculer zero –v1
  - Calculer max (v1, zero – v1)

Transferts mémoire

- Transferts flottants
  - Transferts entre registres XMMi et Mémoire
    - movaps xmm1, [eax]
    - movaps [edi], xmm2
  - Transferts alignés ou non
    - Aligné 4 mots : movaps
    - Aligné 2 mots haut : movhps
    - Aligné 2 mots bas : movlps
    - Non aligné : movups
  - Transfert scalaire
    - movss : charge mot bas du registre, et met à 0 les autres
  - Transfert entre partie haute ou basse de registres
    - movhlps et movlhps
- Transferts entiers
  - Transfert entre mémoire et registres XMM

Alignement mémoire

- L'accès à un mot de 128 bits est aligné sur une frontière de 16 octets. Les quatre bits de poids faible de l'adresse sont 0000 pour un accès aligné
- IA-32
  - Accès alignés
  - Accès non alignés (plus lents)
- Altivec
  - Les accès mémoire sont obligatoirement alignés
  - Utilisation de plusieurs instructions pour les accès non alignés

Accès mémoire (IA-32)

- Accès mémoire “non write allocate”
  - movntps : XMM vers mémoire
  - movntq : MMX vers mémoire
  - sur un défaut de cache en écriture, il y a écriture directe en mémoire
- Préchargement
  - prefetcht0 : L1 et L2
  - prefetcht1 et prefetcht2 : L2 seul
  - prefetchnta : L1 seul

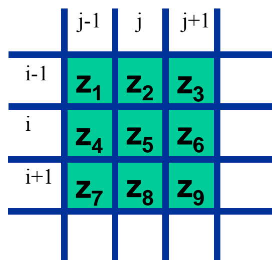
Problèmes d’alignement

- Variables allouées statiquement
  - __declspec(align(16)) V1, V2, V3;
- Variables allouées dynamiquement
  - #include malloc.h
  - Fonctions _mm_malloc et _mm_free

```
byte** bmatrix(long nrl, long nrh, long ncl, long nch)
{long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
 byte **m;
 /* allocate pointers to rows */

 m=(byte **) _mm_malloc((size_t)((nrow+NR_END)*sizeof(byte*)), 16);
 if (!m) nrerror("allocation failure 1 in bmatrix()");
 m += NR_END;
 m -= nrl;
/* allocate rows and set pointers to them */
……..
   return m;
```

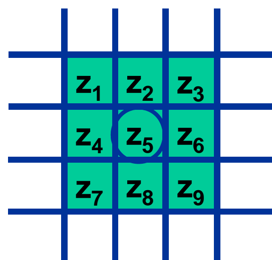
Problèmes d'alignement : les filtres (3,3)

- Traitement d'images bas niveau
  - Filtres passe-bas
  - Filtres passe-haut
  - Etc.



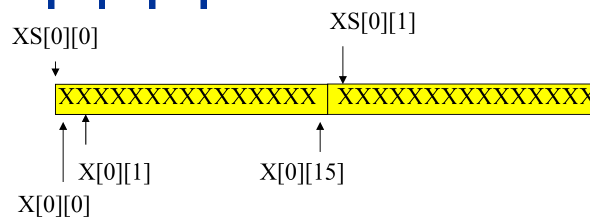
- Y[i][j] = fonction (X[i-1][j-1], X[i-1][j], X[i-1][j+1], X[i][j-1], X[i][j], X[i][j+1], X[i+1][j-1], X[i+1][j], X[i+1][j+1]);
- unsigned char Y[N][N], X[N][N]

Accès aux pixels voisins

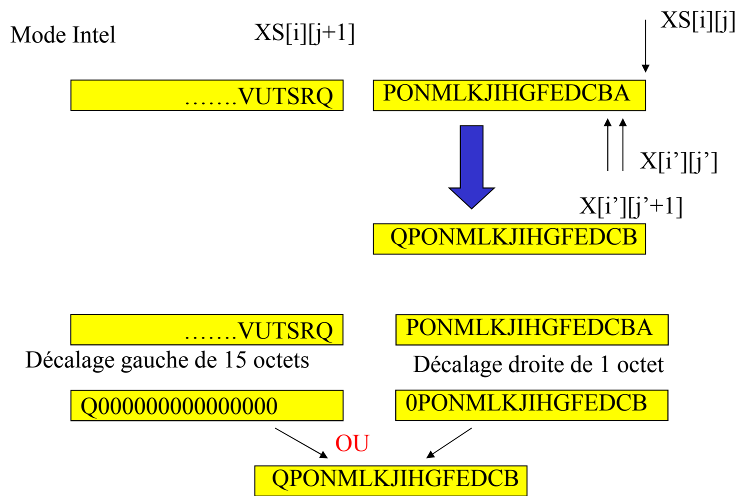



```
byte **X[i][j];
__m128i **XS [i][j];

XS = X;
```

Accès classique ≠ accès Intel



Accès SIMD aux pixels voisins



Accès aux pixels voisins

```
#define decg(va,vb)  _mm_or_si128 (_mm_srli_si128(va,1),_mm_slli_si128(vb,15))
#define decd(va,vb)  _mm_or_si128 (_mm_slli_si128(va,1),_mm_srli_si128(vb,15))
```

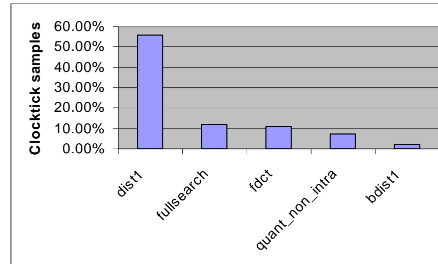
EXEMPLE

```
aij= _mm_load_si128(&XS[i][j]);

aijp = decg(_mm_load_si128(&XS[i][j]),_mm_load_si128(&XS[i][j+1]));

aijm= decd(_mm_load_si128(&XS[i][j]),_mm_load_si128(&XS[i][j-1]));
```

Instructions spéciales (ex : PSABW)

- PSABW
  - Valeur absolue des différences des octets (unsigned char) dans DST et SRC
  - Somme des valeurs absolues pour les huit octets bas dans les 32 bits de la partie basse de DST
  - Somme des valeurs absolues pour les huit octets hauts dans les 32 bits de la partie haute de DST

0000	RES2	0000	RES1



Encodeur MPEG2

$$SAE = \sum_{i=0}^{i=7}\sum_{j=0}^{j=7}\left|C_{ij} - R_{ij}\right|$$

Impact de PSABW

Code C pour l'estimation de mouvement

```
for(l=0; l<nVERT; l++)
   for(k=0, c=0; c<nHORZ; k=+8, c++){
        answer = 0;
        for(j=0; j<16; j++)
           for(i=0; i<16; i++)
              answer += abs(x[l+j][k+i] - y[l+j][k+i]);
       result[l][c] = answer;}
```

Version C naïve : 271 CPP (cycles par pixel)
Version XMM : 13,5 CPP
Accélération : **20**

ALTIVEC (PowerPC)

- Intégré à un jeu d'instructions RISC (PowerPC)
- Unité de traitement SIMD
  - 32 registres vectoriels
  - 160 instructions vectorielles
- Opérandes
  - 16 octets
  - 8 entiers 16 bits
  - 4 entiers 32 bits
  - 4 flottants simple précision
- Format d'instructions

Code op	RD	RS1	RS2	FILTRE

ALTIVEC : addition vectorielle

- Existe dans tous les formats entiers et flottant.

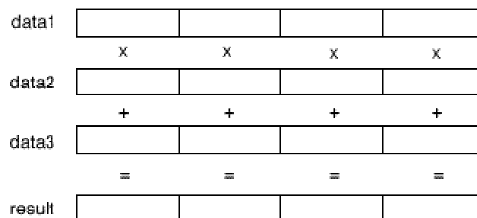
```
vaddubm result, data1, data2
```

data1 [16 cases]
\+ + + + + + + + + + + + + + + +
data2 [16 cases]
= = = = = = = = = = = = = = = =
result [16 cases]

ALTIVEC : multiplication accumulation

- Multiplie quatre “floats” par quatre “floats” et ajoute à quatre autres “floats” en une instruction `vmaddfp`:

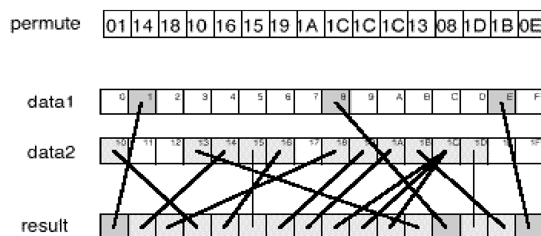
```
vmaddfp result, data1, data2, data3
```



ALTIVEC : instruction permute

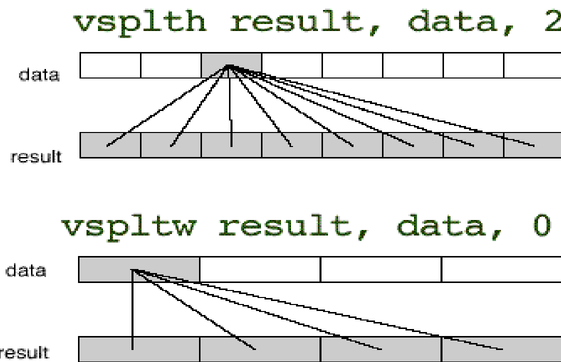
- L’instruction “permute” remplit un registre à partir de deux autres registres. Les octets peuvent être spécifiés dans n’importe quel ordre.

```
vperm result, data1, data2, permute
```

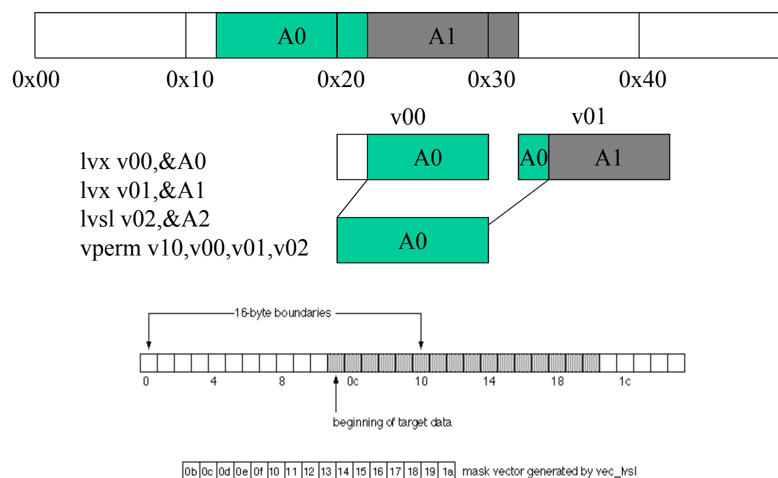


ALTIVEC: instruction SPLAT

- L'instruction “splat” est utilisée pour copier un élément d'un registre dans tous les éléments d'un autre registre



Accès mémoire Altivec non alignés



Les problèmes d'utilisation des instructions SIMD (« vectorisation »)

- **Les obstacles intrinsèques à la vectorisation**
  - Problèmes fondamentaux de vectorisation/parallélisation
  - Ex : les dépendances de données entre itérations
- **Ce qui empêche le compilateur de « vectoriser »**
  - Pointeurs
  - Accès aux tableaux avec pas non unitaires…
  - Structures
  - Etc…
- 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 IA-32
  - Problèmes d'alignement mémoire
  - Manque de certaines instructions (« réduction »)

Problèmes intrinsèques : 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

Eviter les pointeurs

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

```
int a[100];
int *p;
p=a;

for (i=0; i<100;i++)
   *p++ = i;
```

Ne vectorise pas

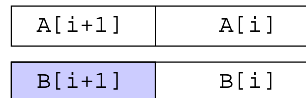
```
int a[100];

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

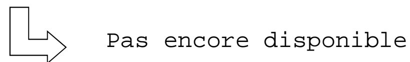
VECTORISE

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

Echange 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++) {
            a[i,j] = a[i,j]+ b[i,j]*c[i,j];}}}
```

NV

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++) {
            a[i,j] = a[i,j]+ b[i,j]*c[i,j];}}}
```

V

PAS ==1

Problèmes intrinsèques : 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

Eclatement de boucles

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

NON VECTORISABLE

```
for(int i=0; i<99; i++) {
          d[i] = e[i] * k[i];}
```

VECTORISABLE

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

NON VECTORISABLE

Fusion de boucles

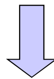
```
void loop_fusion_example(float *a, float *b, float *c, float *d)
{ for(int i=0; i<100; i++)          V
      a[i] = b[i] + c[i];
  for(i=0; i<99; i++)              V
      d[i] = a[i] * 2;}

 for(i=0; i<99; i++) {             V
      a[i] = b[i] + c[i];
      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];}
```

NON VECTORISABLE



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

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

FFT : Version originale

```
void fft_c(int n,COMPLEX *x,COMPLEX *w)
{COMPLEX u,temp,tm;
int i,j,le,windex;
windex = 1;
   for(le=n/2 ; le > 0 ; le/=2) {
      wptr = w;
      for (j = 0 ; j < le ; j++) {
         u = *wptr;
         for (i = j ; i < n ; i = i + 2*le) {
            xi = x + i;
            xip = xi + le;
            temp.real = xi->real + xip->real;
            temp.imag = xi->imag + xip->imag;
            tm.real = xi->real - xip->real;
            tm.imag = xi->imag - xip->imag;
           xip->real = tm.real*u.real - tm.imag*u.imag;
           xip->imag = tm.real*u.imag + tm.imag*u.real;
            *xi = temp; }
         wptr = wptr + windex;}
      windex = 2*windex;}}
```

- Embree, C algorithms for Real-Time DSP, Prentice Hall
- Caractéristiques
  - POINTEURS
  - STRUCTURE
  - PAS NON UNITAIRES
- Caractéristiques qui empêchent le compilateur de vectoriser
  - POINTEURS
  - STRUCTURE
  - PAS NON UNITAIRE

FFT - 1) Supprimer les pointeurs

```
void fft_c(int n,COMPLEX *x,COMPLEX *w)
{
   COMPLEX u,temp,tm;
          int i,j,le,windex;
windex = 1;
   for(le=n/2 ; le > 0 ; le/=2) {
      for (j = 0 ; j < le ; j++) {
         k=0;
         for (i = j ; i < n ; i = i + 2*le) {
            tm.real = x[i].real - x[i+le].real;
            tm.imag = x[i].imag - x[i+le].imag;
            x[i+le].real = tm.real*w[k].real - tm.imag* w[k].imag;
            x[i+le].imag;= tm.real*w[k].imag + tm.imag* w[k].real;
            x[i].real = x[i].real + x[i+le].real;
            x[i].imag = x[i].imag + x[i+le].imag; }
         k += windex;
      }
      windex = 2*windex }
```

STRUCTURE
PAS NON UNITAIRE

En compilation, domaine de recherche actif : *transformation des accès via pointeurs en accès directs via indices de tableaux.*

FFT : 2) Permutation de boucles + Suppression de la structure

```
void fft_c(int n, float x_r[],float x_i, float w_r[], float w_i[])
{
        register float tm_r, tm_i;
        int i,j,le,windex;
windex = 1;
  for(le=n/2 ; le > 0 ; le/=2) {
        for (i = 0 ; i < n ; i = i + 2*le) {
            for (j = 0 ; j < le ; j++){
                tm_r = x_r[i+j] – x_r[i+j+le];
                tm_i = x_i[i+j]– x_i[i+j+le];
                x_r[i+j] = x_r[i+j] + x_r[i+j+le];
                x_i[i+j] = x_i[i+j] + x_i[i+j+le];
                x_r[i+j+le] = tm_r*w_r[j*windex] – tm_i* w_i[j*windex];
                x_i[i+j+le] = tm_r*w_i[j*windex] + tm_i* w_r[j*windex];
            }
        }
    windex = 2*windex;
  }
}
```

PAS NON UNITAIRE POUR L'ACCES AUX COEFFICIENTS

FFT – 3) Programme vectorisé

```
void fft_c(int n, float x_r[],float x_i[],float
w_r[],float w_i[])
{
  register float t_r,t_i;
  int i,j,le,windex, k;
  windex = 1;k=0;
  for(le=n/2 ; le >0 ; le/=2,k++) {
    for (i = 0 ; i < n ; i = i + 2*le) {
      for (j = 0 ; j < le ; j++) {
          t_r=x_r[i+j]-x_r[i+j+le];
          t_i=x_i[i+j]-x_i[i+j+le];
          x_r[i+j]=x_r[i+j]+x_r[i+j+le];
          x_i[i+j]=x_i[i+j]+x_i[i+j+le];
          x_r[i+j+le]=t_r*w_r[HN*k+j]-
                  t_i*w_i[HN*k+j];
          x_i[i+j+le]=t_r*w_i[HN*k+j]+
                  t_i*w_r[HN*k+j];}}
    windex = 2*windex; }
```

Calcul des coefficients

```
for(i = 0 ; i < n ; i++) {
    wr[i] = cos(i*a);
    wi[i] = sin(i*a);}
        windex = 1;k=0;
for(le=n/2 ; le > 0 ; le/=2) {
    for (j = 0 ; j < le ; j++) {
        w_r[HN*k+j]= wr[j*windex];
        w_i[HN*k+j]= wi[j*windex];}
    k++;
    windex = 2*windex;
  }
```

Pour les deux dernières boucles externes, moins de 4 boucles internes

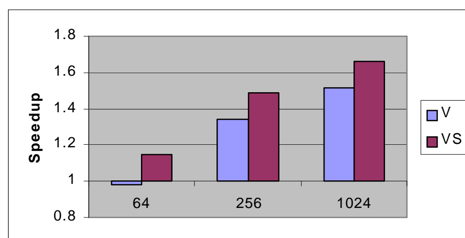
FFT : 4) Programme final vectorisé

```
void fft_c(int n, float x_r[],float x_i[],float
w_r[],float w_i[])
{
  register float t_r,t_i;
  int i,j,le,windex, k;
   windex = 1;k=0;
   for(le=n/2 ; le > 2 ; le/=2,k++) {
      for (i = 0 ; i < n ; i = i + 2*le) {
        for (j = 0 ; j < le ; j++) {
              t_r=x_r[i+j]-x_r[i+j+le];
              t_i=x_i[i+j]-x_i[i+j+le];
              x_r[i+j]=x_r[i+j]+x_r[i+j+le];
              x_i[i+j]=x_i[i+j]+x_i[i+j+le];
              x_r[i+j+le]=t_r*w_r[HN*k+j]-
                          t_i*w_i[HN*k+j];
              x_i[i+j+le]=t_r*w_i[HN*k+j]+
                          t_i*w_r[HN*k+j];}}
      windex = 2*windex; }
```

```
for( ; le > 0 ; le/=2,k++) {
      for (i = 0 ; i < n ; i = i + 2*le) {
            #pragma novector
      for (j = 0 ; j < le ; j++) {
            t_r=x_r[i+j]-x_r[i+j+le];
            t_i=x_i[i+j]-x_i[i+j+le];
            x_r[i+j]=x_r[i+j]+x_r[i+j+le];
            x_i[i+j]=x_i[i+j]+x_i[i+j+le];
            x_r[i+j+le]=t_r*w_r[HN*k+j]-
                        t_i*w_i[HN*k+j];
            x_i[i+j+le]=t_r*w_i[HN*k+j]+
                        t_i*w_r[HN*k+j];}
      }
      windex = 2*windex;
   }
}
```


Accélération FFT

Pentium 4 – 1,4 GHz – 512 Mo RDRAM
Compilateur Intel C++ Beta 6.0



- Performances obtenues inférieures à celles de la bibliothèque Intel
- Version finale est contraire aux « habitudes» des programmeurs C
  - Pointeurs
  - Structure
- Version finale est contraire à ce qui est enseigné :
  - « Use post-incremented pointers to access data in arrays rather than subscripted variables (x=array[i++] is slow, x=*ptr++ is faster) »
  - Embree, C algorithms for Real-Time DSP.

Les problèmes d'utilisation du SIMD

- Les obstacles intrinsèques à la vectorisation
  - Problèmes fondamentaux de vectorisation/parallélisation
  - Ex : les dépendances de données entre itérations
- Ce qui empêche le compilateur de « vectoriser »
  - Pointeurs
  - Accès aux tableaux avec pas non unitaires…
  - Structures
  - Etc…
- **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 IA-32
  - Problèmes d'alignement mémoire
  - Manque de certaines instructions (« réduction »)

A nouveau sur les filtres (images)

- Caractéristiques
  - Stockage des images (octets)
  - Traitement (shorts ou int)
  - Résultat (octets)
- Exemples : filtres, gradient, scan

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);   } }
```

Et solution : Deriche HV

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 flottant

Deriche entier : 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);}}/
```

Version HV

-Pas de dépendances

-Pas unitaire

Mais

8 bits et *32 bits*

8 bits et 16 bits

Deux problèmes

- formats différents
- multiplication SIMD : 16 x 16 ➔ 32 bits
  - 16 x 16 ➔ partie basse des 32 bits
  - 16 x 16 ➔ partie haute des 32 bits
  - Vectorisation automatique impossible
  - Intrinsics ou assembleur éventuellement

Deriche entier (gradient) : formats différents

```
void INT_Gradient_C_B(byte **m, int size)
{int i, j;  byte **X, **G; …
for(i=0; i<size-1; i++) {
    for(j=0; j<size-1; j++) {
            G[i][j] = (byte) (abs(-X[i][j]+X[i][j+1] - X[i+1][j]+X[i+1][j+1])
                              + abs(-X[i][j]-X[i][j+1] +X[i+1][j] + X[i+1][j+1]));}}
```

> 255

Changement de format lors des calculs intermédiaires

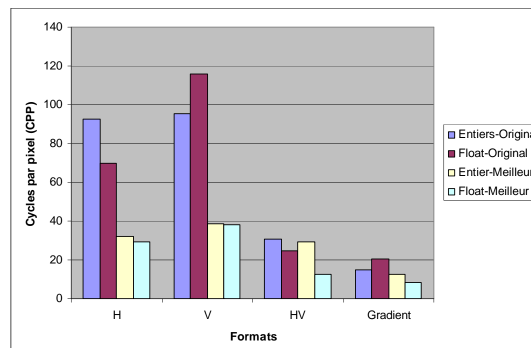
- •Formats identiques
  - •Connaissance de l'amplitude des valeurs
  - •Perte de précision
- •Formats différents
  - •Pas de vectorisation automatique
  - •Vectorisation « éventuelle » à la main

Entiers ou flottants ?

- Problème des calculs intermédiaires en entiers
- Représentation flottante
  - Coût de mémorisation des pixels (8 bits – 32 bits)

- Dell Pentium 4 mobile à 1,6 GHz, 128 Mo RAM
- Windows XP
- Compilateurs Intel C/C++ version 5

Filtres et gradient



Structure de données : AoS, SoA, ou HSoA

- array_of_struct (AoS)
- struct_of_array (SoA)
- Hybrid SoA - Array of SoA

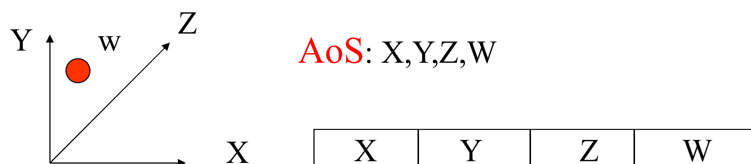
```
struct {
   float  x, y, z, r, g, b;
} AoS_xyz_rgb[200];
```

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

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


Représentation des vertices (AOS-SOA)

Calcul géométrique 3D



X	Y	Z	W

SoA

Tableau Vx	X1	X2	X3 …. Xn
Tableau Vy	Y1	Y2	Y3 …. Yn
Tableau Vz	Z1	Z2	Z3 …. Zn
Tableau Vw	W1	W2	W3 ….Wn

Calcul du produit scalaire

AoS

X	Y	Z	W

mulps ;	a=x*x' b =y*y' c= z*z'
andps;	avec masque pour que d=0
movaps;	transfert reg à reg
shufufps;	donne b, a, d, c à partir de a, b, c, d
addps;	donne a+b, a+b, c+d, c+d
movaps;	transfert registre à registre
shufps;	donne c+d, c+d, a+b, a+b
addps;	donne a+b+c+d, a+b+c+d a+b+c+d a+b+c+d

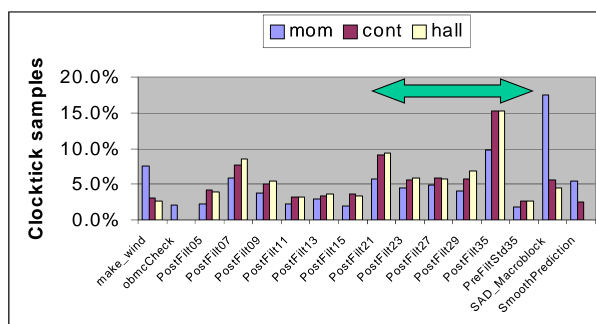
SoA

X	Y	Z	W

mulps ;	x*x' pour les composantes x de 4 sommets
mulps ;	y*y' pour les composantes y de 4 sommets
mulps ;	z*z' pour les composantes z de 4 sommets
addps ;	x*x'+y*y'
addps ;	x*x'+y*y'+z*z'

A NOUVEAU SUR LE PRODUIT SCALAIRE

Codec "Matching Pursuit Video" de Berkeley



CODEUR

FONCTIONS CRITIQUES

SAD (DIST1)

Fonctions Post-filter

Fonctions Post-filter = produits scalaires

Calculs flottants

```
void postfiltn (//……..//)
{float *basis;  *ppm, *ppm_end;
for(ppm=premult+bshift,y=0; y<sr; ppm+=ppm_yinc, ++y) {
   for (ppm_end = ppm+((tmp3<sr)?tmp3:sr);
        ppm < ppm_end;  ++ppm,  --tmp3){
      ip = 0;
      for (k=0; k<n; k++)
          ip+= basis[k]*ppm[k];…..}
  // two similar "middle" loops}
```

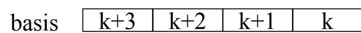
OPTIMISATIONS

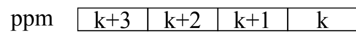
- Déroulage de la boucle interne
  - Calculer plus vite chaque produit scalaire
- Déroulage de la boucle milieu
  - Calculer quatre produits scalaires simultanément

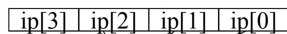
Instructions SIMD pour le produit scalaire

```
for (ppm_end = ppm+((tmp3<sr)?tmp3:sr);
        ppm < ppm_end;  ++ppm,  --tmp3){
   ip[0] = 0.0; ip[1] = 0.0; ip[2] = 0.0; ip[3] = 0.0;
   for (k=0; k<n/4; k+=4){
        ip[0]+= basis[k]*ppm[k];
        ip[1]+= basis[k+1]*ppm[k+1];
        ip[2]+= basis[k+2]*ppm[k+2];
        ip[3]+= basis[k+3]*ppm[k+3]; }
   ip= ip[0]+ip[1]+ip[2]+ip[3];
```

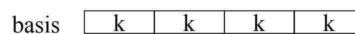
```
for(ppm_end=ppm+((tmp3<sr)?tmp3:sr);
ppm<ppm_end-4; ppm+=4, tmp3-=4){
…..
   ip0 = 0; ip1 = 0; ip2 = 0; ip3 = 0;
   for (k=0; k<n; k++){
     ip0+= basis[k]*ppm[k];
     ip1+= basis[k]*ppm[k+1];
     ip2+= basis[k]*ppm[k+2];
     ip3+= basis[k]*ppm[k+3]; }  …}
```

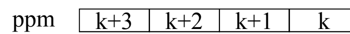
basis 

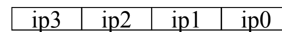
ppm 



Problème d'alignement
Somme finale

basis 

ppm 

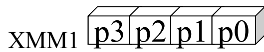


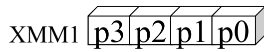
Pas de problème d'alignement
Pas de somme finale

Surcoût de la somme intra-registre (SSE2)

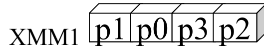
Shuffle

shufps xmm1, xmm1, imm8
0100 1110

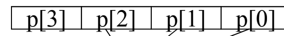
XMM1 

XMM1 

10 11 11 00

XMM1 

Somme finale du produit scalaire



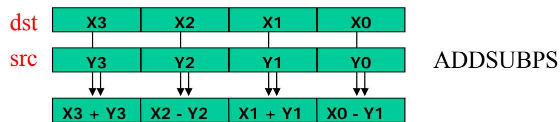
ip

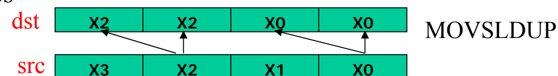
```
movaps xmm1,xmm0        6 clocks
shufps xmm1,xmm1,4eh    6
addps xmm0,xmm1         4
movaps xmm1,xmm0        6
shufps xmm1,xmm1,11h    6
addss xmm0,xmm1         4
movss p,xmm0            4+
```

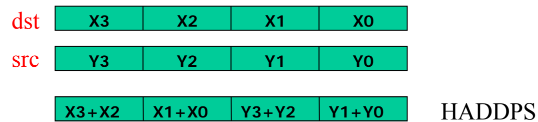
36+ clocks

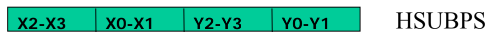
Extensions SIMD SSE3 (IA-32)

- Instructions pour la FFT
  - ADDSUBPD
  - ADDSUBPS
- Duplication de constantes
  - MOVDDUP
  - MOVSHDUP
  - MOVSLDUP
- Instructions horizontales
  - HADDPD
  - HADDPS
  - HSUBPD
  - HSUBPS
- Accès non alignés
  - LDDQU









Réduction des éléments d'un vecteur

- Instruction HADDPS

dst	X3	X2	X1	X0
src	Y3	Y2	Y1	Y0
dst	X3+X2	X1+X0	Y3+Y2	Y1+Y0

XMM0	D	C	B	A
HADDPS XMM0, XMM0	C+D	A+B	C+D	A+B
HADDPS XMM0, XMM0	A+B+C+D	A+B+C+D	A+B+C+D	A+B+C+D

Jeu d'instructions MIC
Co-processeur Xeon Phi

Daniel Etiemble
de@lri.fr

Registres

- 32 registres 512 bits (Zmm0 à Zmm31)
  - 16 int32
  - 8 int64
  - 16 float
  - 8 double
- 8 registres de masque vectoriel k0 à k15 (16 bits)
  - 1 bit par élément du registre Zmm_i (16 ou 8 bits)
    - Contrôle des opérations « élément par élément »
    - Contrôle des écritures registre « élément par élément »
  - Exception
    - k0 correspond au « non masquage »

Instructions vectorielles

- Format général (3,1)
  - $Zmm_i \leftarrow Zmm_j$ OP S (Zmm_k, m)
    - Zmm_i : Registre destination
    - Zmm_j : Registre source 1
    - Zmm_k : Registre source 2
    - m : opérande mémoire
    - S : Fonction de conversion – Swizzle
    - OP : Opération

Addition vectorielle sur des float32

vaddps zmm1 {k1}, zmm2, *S_{f32}(zmm3/m)*

Memory Up-conversion: S_{f32}

$Zmm_i \leftarrow Zmm_j$ OP Opérandes mémoire

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32