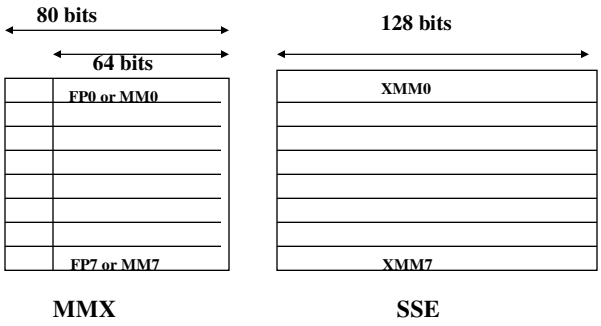
Extensions SIMD dans les microprocesseurs

Daniel Etiemble
de@lri.fr

Extensions SIMD dans les jeux d'instructions

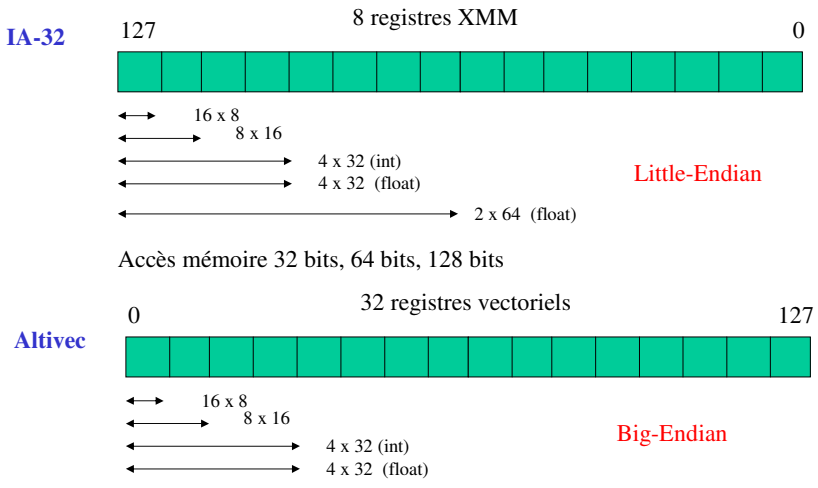
- Dans tous les jeux d'instructions
  - HP, SPARC VIS, MIPS, PowerPC AltiVec
  - IA-32 (Intel MMX, SSE, SSE2,SSE3, AMD 3D Now)
- Applications
  - Audio, Communication, Noyaux DSP, Graphique 2D et 3D, Images, Vidéo, Reconnaissance parole, etc
- Formes limitées d'instructions vectorielles
  - Vecteurs courts (2, 4, 8, 16 éléments)
  - 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)



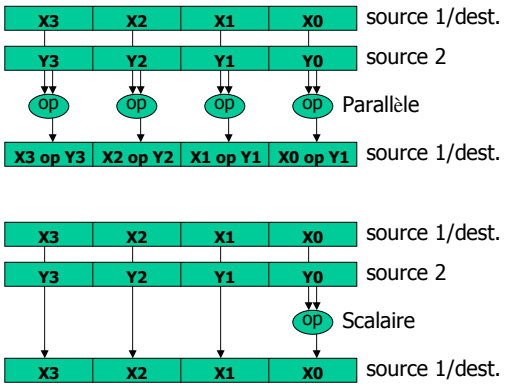
- Les registres MMX sont partagés avec les registres flottants (pile x87)
- Les registres XMM sont distincts

Formats de données SIMD ()



Instructions SIMD parallèles et scalaires

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



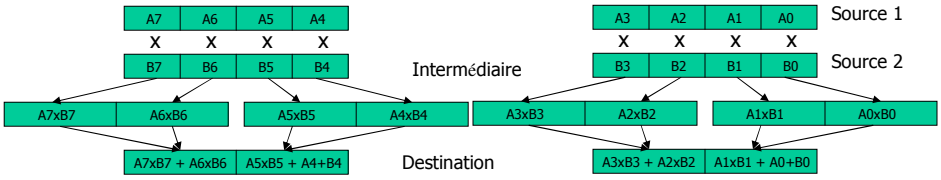
SIMD IA-32 : opérations arithmétiques

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

Multiplication - accumulation entière

- 8 multiplications et 4 additions en une instruction PMADDWD

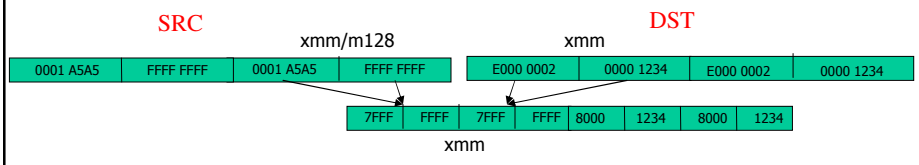


- PMADDWD produit 2 résultats 32 bits
  - Utile pour les applications multimédia et traitement du signal
  - Formats d'entrée et de sortie différents
  - N'existe pas pour les données 8 bits en entrée

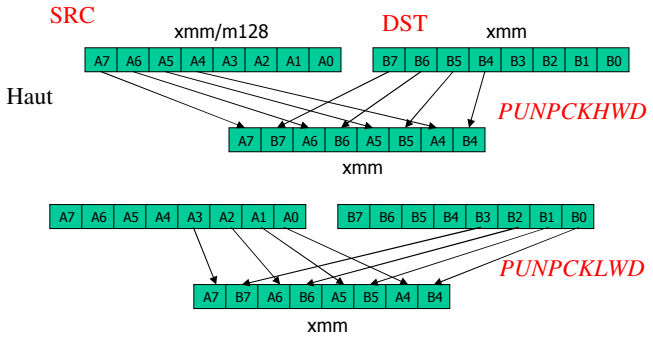
Compactage - décompactage

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



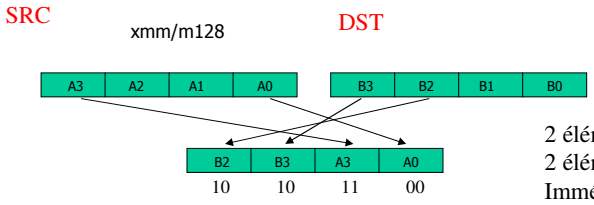
PUNPCKHBW
PUNPCKHWD
PUNPCKHDQ
PUNPCKHDDQ

PUNPCKLHBW
PUNPCKLHWD
PUNPCKLDQ
PUNPCKLDDQ

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

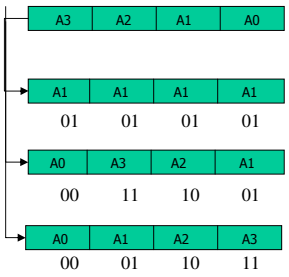
Les instructions "shuffle"

PSHUFD
PSHUFHW
PSHUFLW
SHUFPD
SHUFPS



2 éléments de DEST
2 éléments de SCR
Immédiat 8 bits

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



Transferts mémoire

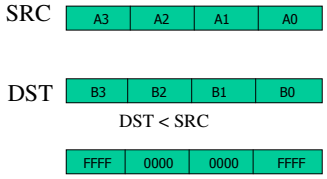
- Transferts flottants
  - Transferts entre registres XMMi et Mémoire
    - movaps xmm1, [eax]
    - movaps [edi], xmm2
  - Transferts alignés ou non
    - Aligné 4 mots : movaps
    - Aligné 2 mots haut : movhps
    - Aligné 2 mots bas : movlps
    - Non aligné : movups
  - Transfert scalaire
    - 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

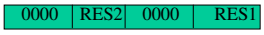
Comparaisons et opérations logiques

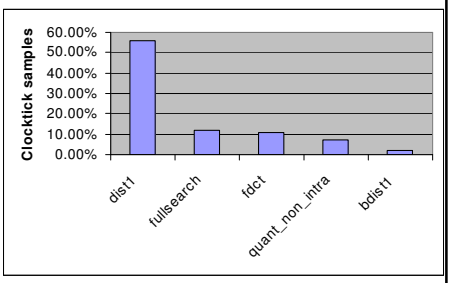
- Compare (eq, lt, le, unord, neq, nlt, nle, ord) and set mask
  - Flottants
    - cmp*xx*ps ou cmp*xx*ss
    - cmp*xx*pd ou cmp*xx*sd
  - Entiers
    - PCOMPEQ (B,W,D)
    - PCOMPGT (B,W,D)
- Opérateurs logiques
  - and, andn, or, xor
    - versions ps et ss
    - Version entière



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





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**

Type d'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)
```

« 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

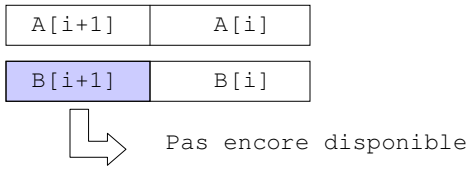
```
int a[100];

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

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

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]; }  }}
```

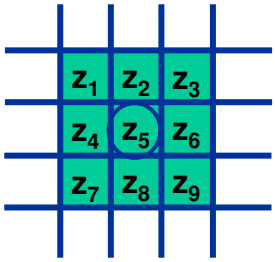
```
void inversionS(byte **X, long i0, long i1, long j0, long j1, byte **Y)
{  int i, j, m, n;
  __m128i **XS, **YS;
  __m128i constante, l1, l2;
 XS=X; YS=Y;
constante=_mm_set1_epi8 (-1);
for(i=i0; i<=i1; i++) {
   for(j=j0; j<=j1/16-1; j++) {
              _mm_store_si128(&YS[i][j], _mm_subs_epu8(constante,XS[i][j]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


Problèmes d'alignement

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

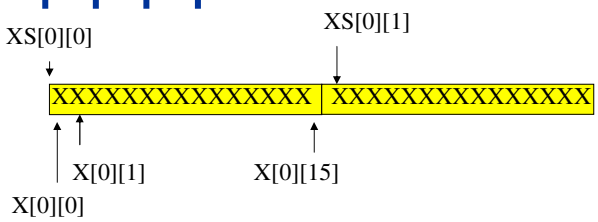
```
byte** bmatrix(long nrl, long nrh, long ncl, long nch)
{long i, nrow=nrh-nrl+1,ncol=nch-ncl+1;
  byte **m;
  /* allocate pointers to rows */

  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;
```

Accès aux pixels voisins

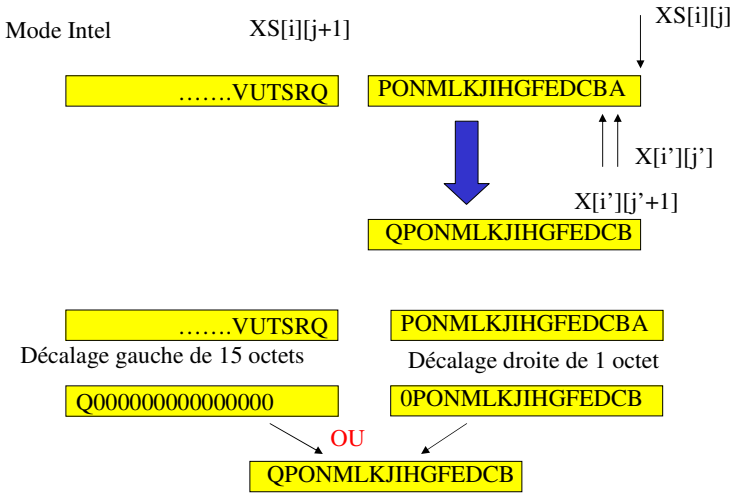



```
byte **X[i][j];
__m128i **XS [i][j];
XS = X;
```

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]));
```


Un exemple avec « produit scalaire »

Calculs flottants

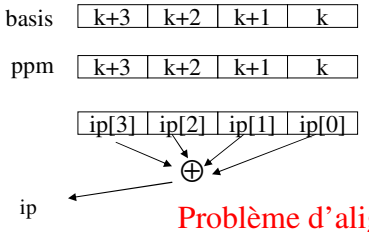
OPTIMISATIONS

```
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}
```

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

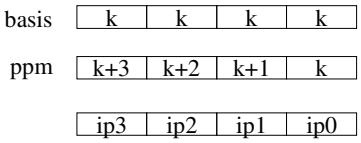
Instructions SIMD et produit scalaire

```
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];
```



Problème d'alignement
Somme finale

```
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]; } …}
```



Pas de problème d'alignement
Pas de somme finale