Parallélisation et Vectorisation automatique

Parallélisation

- Plusieurs coeurs de calcul dans les processeurs actuels,
- Parallélisme de données :
  - ```
    int a[N], b[N];
    for(unsigned i= 0; i < N; i++)
     a[i]= f(b[i]);
    ```
  - ```
    int a[N];
    int m= 0;
    for(unsigned i= 0; i < N; i++)
     // ou un opérateur commutatif et associatif
     m= max(m, a[i]);
    ```

Parallélisation

- Plusieurs coeurs pour paralléliser…
- Et avec un seul ?

MMX, SSE{2,3,4}, AVX, AVX2, AVX512…

- Depuis les pentiums (1995 !) unités de calcul vectorielles,
- Évolution régulière du jeu d’instructions,
- Et de la taille du vecteur !
- SSE : 128 bits, 2 doubles, ou 4 floats, ou 4 ints, ou 8 octets !
- AVX : 256 bits, 4 doubles, 8 floats, 8 floats, ou 16 octets,
- AVX512 : 512 bits, 8 doubles, 16 floats…

- AVX Disponible depuis 2011 sur AMD et Intel
- ARM : NEON 128 bits depuis 2009
- RISC-V : taille variable !

Unité de calcul vectorielle ?

- Comme les unités de calculs classiques,
- mais plusieurs valeurs en même temps !
- Opérateurs arithmétiques, binaires, min, max, etc.
- AVX : 8 floats ou 8 int :
- ```
  __m256i a= { 1, 2, 3, 4, 5, 6, 7, 8 };
  __m256i b= { 10, 10, 10, 10, 10, 10, 10, 10 };
  __m256i c= _mm256_add_epi32(a, b);
  ```
- NEON : au moins 4, sans doute plus, à vérifier (!)
  ```

Unité de calcul vectorielle AVX

- Les instructions sont nommées de manière „créative“ !
- Moteur de recherche quasi-nécessaire : intrinsics-guide,
- Registres 256 bits non typés,
- Opcode de l’instruction indique le type et le nombre de données :

  `_m256_add_ps()` pour additionner 8 floats,
  `_m256_add_epi32()` pour 8 ints 32bits, etc.

Exemple 1

- Maximum d'un ensemble d'entiers :
- ```
  int a[N];
  int m= 0;
  for(unsigned i= 0; i < N; i++)
      m= max(m, a[i]);
  ```
- `_mm256_max_epi32(a, b)`
  renvoie le max de a et b,
- `_mm256_load_si256(__m256i *)`
  pour charger 8 valeurs dans un registre
  (mais l'adresse doit être multiple de 16, trop facile sinon…)
  ```

Exemple 1

```
alignas(16) int t[N];

__m256i m= _mm256_set1_epi32(0);

for(int i= 0; i+7 < N; i+= 8)
{
    __m256i x= _mm256_load_si256((__m256i *) (t + i));
    m= _mm256_max_epi32(m, x);
}
```

Exemple 1

- Il manque un prologue si l'adresse du tableau n'est pas mulitple de 16…

- Et un épilogue si le nombre d'élements n'est pas un multiple de 8 :

```
for(int i= N & ~7; i < N; i++)
    m= max(m, t[i]);
```

Exemple 1

- C’est fini ?
- Pas tout à fait ! Il reste 8 valeurs !
- Une dernière réduction est nécessaire…
- Solution „habituelle“, extraire les 8 valeurs une par une :

```
alignas(16) int tmp[8];
_mm256_store_si256((__m256i *) tmp, m);

int mm= tmp[0];
for(int i= 1; i < 8; i++)
    mm= max(mm, tmp[i]);
```

Exemple 1 : réduction horizontale

Mais c'est quand même moche, et c'est linéaire,
peut mieux faire !

- `_mm256_shuffle_epi32( )` mélange les composantes (attention aux contraintes…)
- `1 2 3 4 5 6 7 8`
  `2 1 4 3 6 5 8 7` `shuffle(2, 3, 0, 1)`
- `2 2 4 4 6 6 8 8` `max()`
  `4 4 2 2 8 8 6 6` `shuffle(0, 1, 2, 3)`
- `4 4 4 4 8 8 8 8` `max()`
  `x x x x 4 4 4 4` `extract(1)`
- `x x x x 8 8 8 8` `max()`
- 3 „mélanges“ et 3 max, et surtout sans stockage en mémoire…

Exemple 1 : réduction horizontale

```
constexpr int shuffle( const int a, const int b, const int c, const int d )  {
        return ((a & 3) << 6) | ((b & 3) << 4) | ((c & 3) << 2) | (d & 3);  }


__m256i x= _mm256_shuffle_epi32( mm, shuffle(2, 3, 0, 1) );
mm= _mm256_max_epi32(mm, x);

__m256i y= _mm256_shuffle_epi32( mm, shuffle(0, 1, 2, 3) );
mm= _mm256_max_epi32(mm, y);

__m256i z= _mm256_castsi128_si256( _mm256_extracti128_si256(mm, 1) );
mm= _mm256_max_epi32(mm, z);

int m= _mm256_cvtsi256_si32(mm);
```

Exemple 1, et alors ?

- „Super !“
  „Et sinon, il y a un truc utilisable ?“

- Oui !
  les compilateurs font cette optimisation automatiquement !
- Dans certains cas (cf opérateur associatif et commutatif)…
- Inspecter le résultat avec : `objdump -d -C -M x86-64 exe`
  ou compiler explorer

clang++ -O3 march=x86-64-v3

```
alignas(16) int a[N];
int m= 0;
for(unsigned i= 0; i < N; i++)
                                    m= max(m, a[i]);
```

```
.LBB0_4:
        vpmaxsd 128(%rsp,%rax,4), %ymm3, %ymm3
        vpmaxsd 160(%rsp,%rax,4), %ymm0, %ymm0
        vpmaxsd 192(%rsp,%rax,4), %ymm1, %ymm1
        vpmaxsd 224(%rsp,%rax,4), %ymm2, %ymm2
        addq    $32, %rax
        cmpq    %rax, %r13
        jne     .LBB0_4

        vpmaxsd %ymm0, %ymm3, %ymm0
        vpmaxsd %ymm2, %ymm1, %ymm1
        vpmaxsd %ymm1, %ymm0, %ymm0
        vextracti128 $1, %ymm0, %xmm1
        vpmaxsd %xmm1, %xmm0, %xmm0
        vpshufd $238, %xmm0, %xmm1
        vpmaxsd %xmm1, %xmm0, %xmm0
        vpshufd $85, %xmm0, %xmm1
        vpmaxsd %xmm1, %xmm0, %xmm0
```

Vectorisation automatique

- Options nécessaires :
  - Optimisation `-O2` ou `-O3`
  - Jeu d'instruction „moderne“ AVX, AVX2 :
    `-mavx2` ou `-march=x86-64-v3`, ou `-march=native`
  - Pour les calculs sur les réels :
    forcer associativité et commutativité sur les float / double
    `-ffast-math`
    compiler explorer

Vectorisation automatique

- Toutes les boucles ne sont pas vectorisables !
- Rappel : parallélisme de données !
- Les itérations doivent être indépendantes,
- Le nombre d’itérations doit être connu,

Exemple 2

- Parfois, il est possible de re-écrire une boucle pour „casser“ les dépendances entre les itérations, ou de rendre le nombre d’itérations explicite :
-

```
unsigned matrices[32];

unsigned sample( unsigned index )
{
    unsigned x= 0;
    for(unsigned i= 0; index; index >>= 1, i++)
        if(index & 1)
            x ^= matrices[i];

    return x;
}
```

Exemple 2

- compiler explorer

```
alignas(16) unsigned matrices[32];

unsigned sample32( const unsigned index )
{
    unsigned x= 0;
    for(unsigned i= 0; i < 32; i++)
    {
        if((index >> i) & 1)
            x ^= matrices[i];
    }

    return x;
}
```

Exemple 3

- Rendre les itérations indépendantes :

```
std::array<int, 32> digits( int x, const int base )
{
    std::array<int, 32> digits= {};
    for(int i = 0; i < 32; i++)
    {
        digits[i]= x % base;
        x= x / base;
    }

    return digits;
}
```

Exemple 3

```
constexpr int pow3_tab[]= { 1, 3, 9, 27, 81, 243, 729,
2187, 6561, 19683, 59049, ... };

std::array<int, 32> digits3_tab( int x )
{
    std::array<int, 32> digits= {};
    for(int i = 0; i < 20; i++)
        digits[i]= (x / pow3_tab[i]) % 3;

    return digits;
}
```