Introduction à la programmation SIMD

EI-SE5 – Calcul haute performance

Adrien Cassagne

Le 27 septembre 2022

Table des matières

Objectifs de ce cours

- Présenter les modèles de programmation vectoriel et SIMD
  - Origines et définitions
  - Vectoriel VS SIMD
  - Plusieurs jeux d'instructions
- Plusieurs niveaux de programmation
  - Instructions assembleur
  - Fonctions intrinsèques
  - Wrappeur SIMD
- Revue des principales opérations
  - Opérations mémoires
  - Opérations arithmétiques
  - Opérations binaires
  - Opérations logiques
  - ...

Table des matières

2 Modèles de programmation vectoriel et SIMD

Les supercalculateurs vectoriels

2 Modèles de programmation vectoriel et SIMD

Les supercalculateurs sont les premiers ordinateurs à avoir utilisé le calcul vectoriel pour accélérer les calculs. Dans ce type d'architecture les instructions encodent le nombre d'éléments qu'elles vont traiter.

Les premiers supercalculateurs vectoriels :

- La machine ILLIAC IV de l'Université de l'Illinois en 1972 ($100 \simeq 150$ MFlop/s)
- Le supercalculateur CDC STAR-100 en 1974 (36 MFlop/s)
- Le supercalculateur Cray-1 en 1976 (160 MFlop/s)

Figure: Cray-1 (1976).

Les supercalculateurs vectoriels

Exemple d'instructions scalaires :

```
1  loop:
2    load a, [p_a]
3    load b, [p_b]
4    add c, a, b # c[i] = a[i] + b[i]
5    store [p_c], c
6    add p_a, p_a, 4 # 4 bytes, 32-bit
7    add p_b, p_b, 4 # 4 bytes, 32-bit
8    add p_c, p_c, 4 # 4 bytes, 32-bit
9    add i, i, 1
10 jumple i, 10000, loop
```

Exemple d'instruction vectorielle :

```
1 # c[1..10000] = a[1..10000] + b[1..10000]
2 addv c(1..10000), a(1..10000), b(1..10000)
```

- L'instruction `addv` est décodée (et *fetchée*) une seule fois !
  - — Gains dans les étages du pipeline *fetch* et *decode*
  - — L'instruction est décodée une fois au lieu de 10000 fois
  - — Moins d'instructions (arithmétique de pointeurs plus simple, pas de boucle)
  - — Chaque élément de `a` est calculé en séquentiel dans les deux cas

Single Instruction Multiple Data (SIMD)

Exemple d'instructions scalaires :

```
1  loop:
2    load a, [p_a]
3    load b, [p_b]
4    add c, a, b # c[i] = a[i] + b[i]
5    store [p_c], c
6    add p_a, p_a, 4 # 4 bytes, 32-bit
7    add p_b, p_b, 4 # 4 bytes, 32-bit
8    add p_c, p_c, 4 # 4 bytes, 32-bit
9    add i, i, 1
10 jumple i, 10000, loop
```

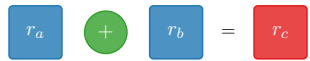
Exemple d'instructions SIMD :

```
1  loop:
2    loadv va, [p_a] # lecture de 4 éléments
3    loadv vb, [p_b] # lecture de 4 éléments
4    addv vc, va, vb # addition sur 4 éléments
5    storev [p_c], vc # écriture de 4 éléments
6    add p_a, p_a, 4*4 # 4*4 bytes, 128-bit
7    add p_b, p_b, 4*4 # 4*4 bytes, 128-bit
8    add p_c, p_c, 4*4 # 4*4 bytes, 128-bit
9    add i, i, 4 # incrémente i de 4
10 jumple i, 10000, loop
```

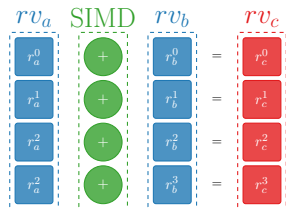
- Les instructions `addv`, `loadv` et `storev` effectuent des opérations sur 4 éléments à la fois
  - Il y a 4 fois moins d'instructions (potentielle accélération de 4)
  - Le nombre d'éléments dans un registre vectoriel = le cardinal (ici 4)
  - Chaque élément de `va` est calculé en parallèle

Single Instruction Multiple Data (SIMD)

- Instruction scalaire: produit une donnée pendant 1 cycle (pour simplifier)



- Une instruction SIMD produit n données pendant 1 cycle (pour simplifier)



- SIMD = *Single Instruction Multiple Data*
- Les instructions SIMD opèrent sur des registres dit “vectoriels” (rv_a, rv_b, rv_c)

Instructions vectorielles VS instructions SIMD

Instructions vectorielles :

- Taille d'une opération définie par l'utilisateur
- Pas de registres vectoriels
- Implémentées dans les vieux supercalculateurs et certains nouveaux (SVE, Fugaku)

Instructions SIMD :

- Taille d'une opération fixe pour un processeur donné
- Présence de registres vectoriels
- Implémentées dans la plupart des ordinateurs ainsi que les supercalculateurs actuels (AVX, NEON)

Dans la suite de ce cours, nous allons nous focaliser sur les instructions SIMD car elles sont plus répandues dans les architectures des processeurs actuels.

Quelques jeux d'instructions SIMD et vectoriels

Date	ISA	Type	Micro-architecture	Taille des registres
1997	MMX	SIMD	Intel Pentium	64-bit
1999	Altivec	SIMD	Apple+IBM PowerPC	128-bit
1999	SSE	SIMD	Intel Pentium III	128-bit
2000	SSE2	SIMD	Intel Pentium IV	128-bit
2004	SSE3	SIMD	Intel Pentium IV (Prescot)	128-bit
2005	NEONv1	SIMD	ARMv7	128-bit
2008	SSE4	SIMD	Intel Core 2 Duo	128-bit
2011	AVX	SIMD	Intel Sandy Bridge (Core i)	256-bit
2012	NEONv2	SIMD	ARMv8	128-bit
2013	AVX2	SIMD	Intel Haswell (Core iX)	256-bit
2016	AVX-512	SIMD	Intel Xeon Phi	512-bit
2017	SVE	Vectoriel	ARMv8	–
2019	SVE2	Vectoriel	ARMv9	–
2021	RVV	Vectoriel	RISC-V	–

Quelques jeux d'instructions SIMD et vectoriels

Date	ISA	Type	Micro-architecture	Taille des registres
1997	MMX	SIMD	Intel Pentium	64-bit
1999	Altivec	SIMD	Apple+IBM PowerPC	128-bit
1999	SSE	SIMD	Intel Pentium III	128-bit
2000	SSE2	SIMD	Intel Pentium IV	128-bit
2004	SSE3	SIMD	Intel Pentium IV (Prescot)	128-bit
2005	NEONv1	SIMD	ARMv7	128-bit
2008	SSE4	SIMD	Intel Core 2 Duo	128-bit
2011	AVX	SIMD	Intel Sandy Bridge (Core i)	256-bit
2012	NEONv2	SIMD	ARMv8	128-bit
2013	AVX2	SIMD	Intel Haswell (Core iX)	256-bit
2016	AVX-512	SIMD	Intel Xeon Phi	512-bit
2017	SVE	Vectoriel	ARMv8	–
2019	SVE2	Vectoriel	ARMv9	–
2021	RVV	Vectoriel	RISC-V	–

- Un grand nombre de jeux d'instructions existants
- Certains sont rétro-compatibles
  - AVX-512 → AVX → SSE → MMX
  - NEONv2 → NEONv1
- La plupart des jeux d'instructions sont incompatibles entre eux
  - Chaque jeux d'instructions s'écrit différemment

Table des matières

3 Plusieurs niveaux de programmation

Assembleur

3 Plusieurs niveaux de programmation

Exemple de l'addition en fonction de différents jeux d'instructions :

x86 SSE

```
1 addps xmm, xmm
```

ARM NEON

```
1 FADD Vd.4S,Vn.4S,Vm.4S
```

x86 AVX

```
1 vaddps ymm, ymm, ymm
```

ARM SVE

```
1 FADDP Zresult.S, Pg.M, Zresult.S, Zop2.S
```

x86 AVX-512

```
1 vaddps zmm, k, zmm, zmm
```

RISC-V Vector extension

```
1 vadd.vv vd, vs2, vs1, vm
```

- Autant de façons différentes que de jeux d'instructions
- Le code assembleur n'est pas portable
- Gros frein pour la productivité

Fonctions intrinsèques : exemple de l'addition

3 Plusieurs niveaux de programmation

x86 AVX

```
1 __m256 _mm256_add_ps (__m256 a, __m256 b);
```

x86 AVX-512

```
1 __m512 _mm512_mask_add_ps (__m512 src, __mmask16 k, __m512 a, __m512 b);
```

ARM NEON

```
1 float32x4_t vaddq_f32 (float32x4_t a, float32x4_t b);
```

ARM SVE

```
1 svfloat32_t svaddp_f32_x (svbool_t pg, svfloat32_t op1, svfloat32_t op2);
```

RISC-V Vector extension

```
1 vfloat32m1_t vfadd_vv_f32m1_m (vbool32_t mask, vfloat32m1_t maskedoff, vfloat32m1_t op1,
2                                vfloat32m1_t op2, size_t vl);
```

Fonctions intrinsèques

3 Plusieurs niveaux de programmation

Définition : une fonction intrinsèque est une fonction à laquelle correspond une instruction assembleur.

- Autant de façons différentes que de jeux d'instructions
- Le code avec des fonctions intrinsèques n'est pas portable
- Meilleure productivité que la programmation assembleur pure
  - Le compilateur s'occupe de l'allocation des registres
  - Simple à interfacer avec du code C et C++
  - Souvent la méthode choisie par les développeurs de systèmes embarqués

S'appuyer sur le compilateur

- Dans certains cas le compilateur est capable de vectoriser automatiquement le code source à votre place
  - Fonctionne quand les calculs sont très réguliers et à partir du niveau d'optimisation `-O3` (GCC)
  - Il est possible d'afficher un rapport de vectorisation pour voir les parties du code qui ont effectivement été vectorisées
- Bien souvent le compilateur ne vectorise pas bien
  - Soit il ne vectorise pas du tout
  - Soit il vectorise un peu mais il pourrait faire beaucoup mieux
- Souvent il est possible d'aider le compilateur avec des annotations (ex: OpenMP SIMD)
- Dans le HPC, beaucoup de codes utilise le compilateur + des annotations
  - Les supercalculateurs évoluent vite
  - Utiliser le compilateur garantit la portabilité de code source

L'alternative : les wrappeurs SIMD

Définition : un wrappeur SIMD est une bibliothèque (C ou C++) qui fait appel aux fonctions intrinsèques ou à de l'assembleur. Le wrappeur SIMD propose une interface de programmation unique pour les différents jeux d'instructions.

- Améliore l'expressivité du code source
  - Possibilité d'utiliser les opérateurs traditionnels (`+`, `-`, `*`, `/`, ...)
- Le code source est portable
  - Il suffit d'écrire le code source une seule fois et il pourra compiler sur plusieurs architectures différentes

C'est le choix fait dans ce cours, nous allons utiliser le wrappeur SIMD "MIPP". C'est un bon compromis entre efficacité et portabilité.

Présentation du wrappeur SIMD “MIPP”

3 Plusieurs niveaux de programmation

- MyIntrinsicsPlusPlus (MIPP), est un code écrit en C++ et qui fait appel aux fonctions intrinsèques
- Développé à l’Inria dans le cadre de la thèse de votre honorable serviteur
- Utilisé dans plusieurs codes de calcul pour bénéficier des instructions SIMD
- Code source ouvert : https://github.com/aff3ct/MIPP
- Exhaustivement testé dans un pipeline d’intégration continue (garanti sans bug, ou presque...)
- Il existe d’autres wrappeurs, mais les mécaniques sont identiques
  - Maîtriser MIPP = faciliter la compréhension des fonctions intrinsèques
  - Maîtriser MIPP = accélérer la prise en main d’autres wrappeurs SIMD

MIPP : Hello World!

3 Plusieurs niveaux de programmation

```
1  void add_vector(const float *A, const float *B, float *C, int size) {
2      // N élements par registre SIMD (statique = connu à la compilation)
3      constexpr int N = mipp::N<float>();
4      // boucle vectorisée
5      for (int i = 0; i < size; i += N) {
6          mipp::Reg<float> rA = &A[i];   // SIMD lecture en mémoire
7          mipp::Reg<float> rB = &B[i];   // SIMD lecture en mémoire
8          mipp::Reg<float> rC = rA + rB; // SIMD addition
9          rC.store(&C[i]);               // SIMD écriture en memoire
10     }
11 }
```

MIPP : Hello World!

3 Plusieurs niveaux de programmation

```
1  void add_vector(const float *A, const float *B, float *C, int size) {
2      // N élements par registre SIMD (statique = connu à la compilation)
3      constexpr int N = mipp::N<float>();
4      // boucle vectorisée
5      for (int i = 0; i < size; i += N) {
6          mipp::Reg<float> rA = &A[i];   // SIMD lecture en mémoire
7          mipp::Reg<float> rB = &B[i];   // SIMD lecture en mémoire
8          mipp::Reg<float> rC = rA + rB; // SIMD addition
9          rC.store(&C[i]);               // SIMD écriture en memoire
10     }
11 }
```

- **Le pas de la boucle est de N !** (le cardinal des registres)

MIPP : Hello World!

3 Plusieurs niveaux de programmation

```
1  void add_vector(const float *A, const float *B, float *C, int size) {
2      // N élements par registre SIMD (statique = connu à la compilation)
3      constexpr int N = mipp::N<float>();
4      // boucle vectorisée
5      for (int i = 0; i < size; i += N) {
6          mipp::Reg<float> rA = &A[i];   // SIMD lecture en mémoire
7          mipp::Reg<float> rB = &B[i];   // SIMD lecture en mémoire
8          mipp::Reg<float> rC = rA + rB; // SIMD addition
9          rC.store(&C[i]);               // SIMD écriture en memoire
10     }
11 }
```

- **Le pas de la boucle est de N !** (le cardinal des registres)
- MIPP travaille au niveau des **registres vectoriels**
  - On peut considérer qu'un `mipp::Reg` est un VRAI registre dans le processeur
  - MIPP est écrit en C++, le code entre les chevrons permet de déterminer le type des éléments à l'intérieur des registres vectoriels (ici `float`)

Table des matières

4 Revue des principales opérations

Registres et mémoire

4 Revue des principales opérations

Définir un registre à partir d'une constante

```
1 mipp::Reg<float> r_cst = 12.f; // r_cst = [12.f, 12.f, 12.f, 12.f]
```

Registres et mémoire

4 Revue des principales opérations

Définir un registre à partir d'une constante

```
1 mipp::Reg<float> r_cst = 12.f; // r_cst = [12.f, 12.f, 12.f, 12.f]
```

Lire des données depuis la mémoire vers un registre

```
1 float array[4] = {0.f, 1.f, 2.f, 3.f};
2 mipp::Reg<float> r_vals = array; // r_cst = [0.f, 1.f, 2.f, 3.f]
```

Registres et mémoire

4 Revue des principales opérations

Définir un registre à partir d'une constante

```
1 mipp::Reg<float> r_cst = 12.f; // r_cst = [12.f, 12.f, 12.f, 12.f]
```

Lire des données depuis la mémoire vers un registre

```
1 float array[4] = {0.f, 1.f, 2.f, 3.f};
2 mipp::Reg<float> r_vals = array; // r_cst = [0.f, 1.f, 2.f, 3.f]
```

Écrire des données depuis un registre vers la mémoire

```
1 mipp::Reg<float> r_vals = {4.f, 5.f, 6.f, 7.f};
2 float array[4] = {0.f, 0.f, 0.f, 0.f};
3 r_vals.store(array); // array = [4.f, 5.f, 6.f, 7.f]
```

Registres et mémoire

4 Revue des principales opérations

Définir un registre à partir d'une constante

```
1 mipp::Reg<float> r_cst = 12.f; // r_cst = [12.f, 12.f, 12.f, 12.f]
```

Lire des données depuis la mémoire vers un registre

```
1 float array[4] = {0.f, 1.f, 2.f, 3.f};
2 mipp::Reg<float> r_vals = array; // r_cst = [0.f, 1.f, 2.f, 3.f]
```

Écrire des données depuis un registre vers la mémoire

```
1 mipp::Reg<float> r_vals = {4.f, 5.f, 6.f, 7.f};
2 float array[4] = {0.f, 0.f, 0.f, 0.f};
3 r_vals.store(array); // array = [4.f, 5.f, 6.f, 7.f]
```

Afficher la valeur d'un registre sur la sortie standard

```
1 mipp::Reg<float> r_vals = {4.f, 5.f, 6.f, 7.f};
2 mipp::dump<float>(r_vals.r); // affiche "[4.f, 5.f, 6.f, 7.f]" à l'écran
```

Opérations arithmétiques usuelles

4 Revue des principales opérations

- MIPP supporte la plupart des opérations arithmétiques usuelles
  - `mipp::add(...)`, `mipp::sub(...)`, `mipp::mul(...)`, `mipp::div(...)`
  - Il est aussi possible de directement utiliser les opérateurs `+`, `-`, `*`, `/` entre deux `mipp::Reg`

```
1 mipp::Reg<float> r_a = {0.f, 1.f, 2.f, 3.f};
2 mipp::Reg<float> r_b = {4.f, 5.f, 6.f, 7.f};
3 mipp::Reg<float> r_c1 = mipp::add(r_a, r_b); // r_c1 = [4.f, 6.f, 8.f, 10.f]
4 mipp::Reg<float> r_c2 = r_a + r_b;           // r_c2 = [4.f, 6.f, 8.f, 10.f]
```

- Utiliser la surcharge d'opérateur est strictement identique à appeler `mipp::add`
- Ne vous en privez pas c'est un des gros avantages des wrappeurs SIMD ;-)

Opérations arithmétiques spécialisées : FMA

- Les jeux d'instructions SIMD sont élaborés pour l'efficacité des calculs
- Plusieurs instruction spécifiques sont implémentées
  - L'opération la plus connu est *Fused Multiply and Add* (FMA) : $d = a \times b + c$.
    - Cette opération est récurrente dans beaucoup d'applications de calcul intensif
    - Dans MIPP : `mipp::fmadd`, `mipp::fnmadd`, `mipp::fmsub`, `mipp::fnmsub`
    - Pas d'opérateur C++ pour la FMA, il faut donc appeler les fonctions MIPP...

```
1 mipp::Reg<float> r_a = {0.f, 1.f, 2.f, 3.f};
2 mipp::Reg<float> r_b = {4.f, 5.f, 6.f, 7.f};
3 mipp::Reg<float> r_c = {1.f, 2.f, 1.f, 2.f};
4 mipp::Reg<float> r_c1 = mipp::fmadd(r_a, r_b, r_c); // r_c1 = [1.f, 7.f, 13.f, 16.f]
5 mipp::Reg<float> r_c2 = r_a * r_b + r_b;            // r_c2 = [1.f, 7.f, 13.f, 16.f]
```

- Attention, les lignes 4 et 5 ne sont pas équivalentes
  - La ligne 4 a un débit d'un cycle alors que la ligne 5 a un débit de 2 cycles
  - À la ligne 4 il y a un seul arrondi tandis qu'à la ligne 5 il y en a deux

Opérations arithmétiques spécialisées : autres

- Il existe beaucoup d'autres opérations spécialisées
  - La racine carré, l'inverse de la racine carré
  - Des produits scalaires pour l'IA (Intel AMX)
  - Calcul de CRC 32-bit
  - Etc.
- Ces fonctions spécialisées varient en fonction des jeux d'instructions
  - Les jeux d'instructions sont très hétérogènes et des choix sont fait en fonction des applications visées
- MIPP supporte une partie de ces opérations spécialisées
  - Voir la documentation : https://github.com/aff3ct/MIPP/blob/master/README.md

Masques et opérations logiques

4 Revue des principales opérations

MIPP supporte la plupart des opérations logiques usuelles :

- `mipp::cmpeq`, `mipp::cmpneq`, `mipp::cmpge`, `mipp::cmpgt`, `mipp::cmple`, `mipp::cmplt`
- Il est aussi possible de directement utiliser les opérateurs `==`, `!=`, `>=`, `>`, `<=`, `<`
- L'utilisation d'opération logique sur des registres vectoriels a pour effet de retourner **un registre de type masque !**

```
1 mipp::Reg<float> r_a = {0.f, 5.f, 2.f, 7.f};
2 mipp::Reg<float> r_b = {4.f, 1.f, 6.f, 3.f};
3 mipp::Msk<mipp::N<float()> r_m = r_a > r_b; // r_m = [0, 1, 0, 1]
```

- Un registre vectoriel de type masque contient l'information *vrai* ou *faux*
- Ici on utilise `mipp::N<float>()` pour récupérer le nombre d'éléments que contient un registre vectoriel de données (`mipp::Reg<float>`)

Opérations bit à bit

4 Revue des principales opérations

MIPP supporte la plupart des opérations bit à bit usuelles :

- `mipp::andb`, `mipp::orb`, `mipp::xorb`, `mipp::lshift`, `mipp::rshift`, `mipp::notb`, ...
- Il est aussi possible de directement utiliser les opérateurs `&`, `|`, `^`, `<<`, `>>`, `~`

Notez que les opérations bit à bit s'appliquent aussi bien aux registres vectoriels de type donnée (`mipp::Reg`) qu'aux registres vectoriel de type masque (`mipp::Msk`).

La condition ternaire ou le *blend* en vectoriel

- Vous l'avez peut-être déjà remarqué : il n'y a pas de condition en SIMD
- Il est cependant possible d'implémenter une condition ternaire

```
1 int res = (val1 > val2) ? val1 : val2; // res = max(val1, val2)
```

La condition ternaire ou le *blend* en vectoriel

- Vous l'avez peut-être déjà remarqué : il n'y a pas de condition en SIMD
- Il est cependant possible d'implémenter une condition ternaire

```
1 int res = (val1 > val2) ? val1 : val2; // res = max(val1, val2)
```

- Il est possible de mimer ce comportement en SIMD

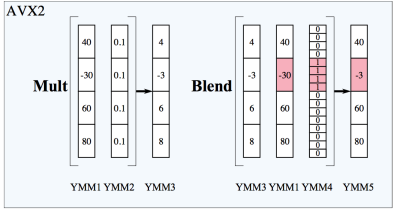
```
1 mipp::Reg<int> r_vals1 = {12, 1, 23, 4};
2 mipp::Reg<int> r_vals2 = {3, 89, 24, 0};
3 mipp::Msk<mipp::N<int>()> r_m = r_vals1 > r_vals2;
4 mipp::Reg<int> r_res1 = mipp::blend(r_vals1, r_vals2, r_m);
5 // r_res1 = {12, 89, 24, 4}
6 mipp::Reg<int> r_res2 = mipp::blend(r_vals1, r_vals2, r_vals1 > r_vals2);
7 // r_res2 = {12, 89, 24, 4}
```

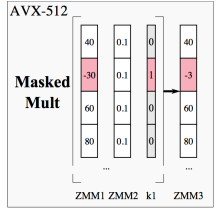
— Les lignes 3-4 ou la ligne 6 sont strictement équivalentes
— La ligne 6 (calcul de `r_res2`) est une version compressée

Jeux d'instructions masqués

4 Revue des principales opérations

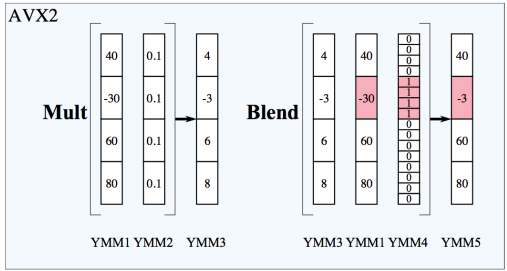
- Combiner une opération arithmétique et un *blend* en un cycle
  - Possible avec AVX-512, SVE et RVV
  - Ne peut pas s'écrire avec un `mipp::blend`
  - Si le jeu d'instructions ne le supporte pas, l'opération arithmétique masqué est émulée (application de l'opération arithmétique puis appel à `mipp::blend`)

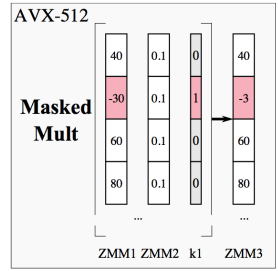




Jeux d'instructions masqués : implémentation

4 Revue des principales opérations






```
1 mipp::Reg<double> r_zmm1 = {40, -30, 60, 80};
2 mipp::Reg<double> r_zmm2 = {0.1, 0.1, 0.1, 0.1};
3 mipp::Msk<mipp::N<double>()> r_k1 = {0, 1, 0, 0}
4 mipp::Reg<double> r_zmm3 = mipp::mask<T,mipp::mul>(r_k1, r_zmm1, r_zmm1, r_zmm2);
5 // r_zmm3 = [40, -3, 60, 80]
```

Shuffles et permutations

La plupart des jeux d'instructions SIMD (et PAS vectoriels) proposent des instructions pour changer l'ordre des éléments au sein d'un même registre vectoriel.

- Être capable de permuter des éléments suppose de connaître le nombre d'éléments (ici on suppose 4 éléments flottants dans un registre vectoriel)

```
1 uint32_t ids[4] = {3, 2, 1, 0};
2 mipp::Reg<float> cmask = mipp::cmask(ids);
3 mipp::Reg<float> r_vals = {10.f, 20.f, 30.f, 40.f};
4 mipp::Reg<float> r_rvals = mipp::shuff(r_vals, cmask);
5 // r_rvals = [40.f, 30.f, 20.f, 10.f]
```

Il arrive assez souvent que l'implémentation de `mipp::shuff` soit assez coûteuse en nombre d'instructions assembleur, cette opération est à utiliser uniquement si nécessaire.

Gather et *scatter*

4 Revue des principales opérations

L'opération *gather* permet de charger des éléments d'un tableau de valeurs en mémoire depuis un registre vectoriel d'indices pour initialiser un registre vectoriel.

```
1 float mem[16] = {0.0f, 10.1f, 20.2f, 30.3f, 40.4f, 50.5f, /*...*/ 150.15f};
2 mipp::Reg<int> r_idx = {0, 5, 12, 2};
3 mipp::Reg<float> r_vals = mipp::gather<float>(mem, r_idx);
4 // r_vals = [0.0f, 50.5f, 120.12f, 20.2f]
```

L'opération *scatter* permet d'écrire les éléments d'un registre vectoriel dans la mémoire. Les adresses d'écriture en mémoire sont déterminées par un registre vectoriel d'indices.

```
1 float mem[16] = {0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, /*...*/ 0.0f};
2 mipp::Reg<int> r_idx = {0, 5, 12, 2};
3 mipp::Reg<float> r_vals = {0.12f, 1.12f, 2.12f, 3.12f};
4 mipp::scatter<float>(mem, r_idx, r_vals);
5 // mem = [0.12f, 0.0f, 3.12f, 0.0f, 0.0f, 1.12f, /*...*/ 0.0f]
```

Réductions ou opérations horizontales

Une réduction (ou une opération horizontale) est une opération qui implique les éléments d'un même registre vectoriel.

- Quelques exemples de réduction
  - La somme des éléments au sein d'un même registre vectoriel
  - Le produit des éléments au sein d'un même registre vectoriel
  - L'élément minimum/maximum au sein d'un même registre vectoriel
  - Etc.
- MIPP vient avec la plupart des réductions usuelles (`mipp::hadd`, `mipp::hmul`, `mipp::hmin`, `mipp::hmax`, `mipp::testz`)

```
1 mipp::Reg<int> r_vals = {1, 2, 3, 4};
2 int sum = mipp::hadd<int>(r_vals); // sum = 10
```

Table des matières

5 Stratégies de vectorisation

Tail loop

5 Stratégies de vectorisation

```cpp
// on suppose que "size = 18"
void add_vector(const float *A, const float *B, float *C, int size) {
    constexpr int N = mipp::N<float>(); // on suppose que "N = 4"
    int vec_loop_end = (size / N) * N; // "vec_loop_end = 16"

    // boucle vectorisée
    for (int i = 0; i < vec_loop_end; i += N) {
        mipp::Reg<float> rA = &A[i];
        mipp::Reg<float> rB = &B[i];
        mipp::Reg<float> rC = rA + rB;
        rC.store(&C[i]);
    }

    // tail loop (scalaire) pour les éléments restants (i = 16 et i = 17)
    for (int i = vec_loop_end; i < size; i++)
        C[i] = A[i] + B[i];
}
```

Lectures et écritures masquées

5 Stratégies de vectorisation

```cpp
// on suppose que "size = 18"
void add_vector(const float *A, const float *B, float *C, int size) {
    constexpr int N = mipp::N<float>(); // on suppose que "N = 4"
    mipp::Reg<int> r_limit = size; // r_limit = [18, 18, 18, 18]
    mipp::Reg<int> r_count = {0, 1, 2, 3};
    // boucle vectorisée
    for (int i = 0; i < size; i += N) {
        mipp::Msk<mipp::N<int>()> r_m = r_count < r_limit;
        // lectures masquées, met un 0 dans le registre si le masque vaut 0
        mipp::Reg<float> rA = mipp::maskzld(r_m, &A[i]);
        mipp::Reg<float> rB = mipp::maskzld(r_m, &B[i]);
        mipp::Reg<float> rC = rA + rB;
        // écriture masquée, n'écrit pas dans la mémoire si l'élément du masque vaut 0
        mipp::maskst(r_m, &C[i], rC);
        r_count += N; // on incrémente le compteur de N
    }
}
```

Q&R

Merci pour votre écoute !
Avez-vous des questions ?