MGP2 – Assembleur et SIMD

Concepts avancés

Instructions avancées

- **_BitScanForward**() / **_BitScanForward64**() retourne l'index du premier bit à 1 en partant des bits de poids faibles.
- **_BitScanReverse**() / **_BitScanReverse64**() retourne l'index du premier bit à 1 en partant des bits de poids forts.
- **__lzcnt16**(), **__lzcnt**(), **__lzcnt64**() « leading-zero count » retourne le nombre de bit a 0 en partant des bits de poids forts.
  - Similaire à _BitScanReverse() mais de façon complémentaire
  - Initialement spécifique aux CPUs AMD, largement supporté de nos jours
- **__popcnt16**(), **__popcnt**(), **__popcnt64**() « population count » retourne le nombre de bits à 1.
  - Initialement spécifique aux CPUs AMD, largement supporté de nos jours

Bit Manipulation Instructions (BMI)

- Les fonctions que l'on vient de voir ne sont pas disponibles sur tous les CPUs x86 / x86_64.
  - L'instruction cpuid permet de connaitre la présence de ces fonctions.

- Ces fonctions sont groupées en ISA comme par exemple ABM (Advanced Bit Manipulation) des processeurs AMD ou BMI1 sur les processeurs AMD (>= Jaguar / Piledriver) ou Intel (>= Haswell)
  - https://en.wikipedia.org/wiki/Bit_Manipulation_Instruction_Sets

- Ce lien présente plusieurs variations intéressantes des fonctions de Bit Scanning https://www.chessprogramming.org/BitScan

Comment simuler un fonctionnement SIMD ?

- Il est possible de simuler le fonctionnement d'un registre SIMD –par exemple un vecteur de 8 bits- via un registre usuel (GPR) de 32 ou 64 bits.

- La solution consiste à s'assurer que les retenues (carry) et les emprunts (borrow) ne se reportent pas sur les bits adjacents.

- Prenons pour exemple la moyenne arrondie de deux entiers A et B:

$$(A + B + 1) / 2$$

  - Une division entière entraîne une perte de perte de résolution qui est corrigée par +1
  - On sait déjà que cette division peut être remplacée par un décalage vers la droite.

Exemple (A+B+1)/2

- Si on applique la loi de distributivité on obtient (A>>1) + (B>>1) + (1>>1) ce qui ne donne plus un résultat correct puisque l'on perd la valeur des premiers bits et du bit correcteur.
  - En fait le (+ 1) est ajouté à la somme des bit 0 (premier bit) de A et B

- Faisons une table de vérité afin d'observer l'état du premier bit

A + B	0	1	A + B + 1	0	1	(A + B + 1)>>1	0	1
0	00	01	0	01	10	0	00	01
1	01	10	1	10	11	1	01	01

- On note que l'opération (A+B+1)>>1 sur **un seul bit** équivaut à un OR
- Appliqué au bit 0 de A et B cela nous donne (A>>1) + (B>>1) + ((A&1) | (B&1))
  - En factorisant (A&1) | (B&1) on obtient la forme (A | B) & 1

SIMD Within A Register (SWAR)

- L'autre problème est que le décalage de A et B entraîne une contamination des bits de poids fort par les bits de poids faible de l'octet de gauche.
- Il suffit ici de masquer chaque octet par 0x7f afin de rétablir une valeur correcte.

R0 = (RA >> 1) & 0x7f7f7f7f
R1 = (RB >> 1) & 0x7f7f7f7f

- De même, on peut appliquer le ET binaire de l'arrondi à chaque octet

R2 = (RA | RB) & 0x01010101

Le résultat de la moyenne arrondie de 4 valeurs en parallèle est donc R = R0 + R1 + R2

Plus loin avec les float-s

Quelques liens sur les réels simple précision

- http://cowboyprogramming.com/2007/01/05/visualizing-floats/
- https://fabiensanglard.net/floating_point_visually_explained
- https://twitter.com/D_M_Gregory/status/1044008750162604032?s=20 (un gif animé très explicite)

- Plusieurs solutions sont possibles, une première consiste à découper le monde en secteurs, la position d'un acteur est donc la position du secteur + la position **relative** de l'acteur dans le secteur (deux float-s). On peut également utiliser des double qui sont souvent aussi rapide que des float sur les architectures modernes.
- Autre alternative, les nombres à virgule fixe (*fixed point*), cf http://tomforsyth1000.github.io/blog.wiki.html#%5B%5BA%20matter%20of%20precision%5D%5D

- Tout ceci devrait nous indiquer que comparer des float est potentiellement hasardeux, il faut donc privilégier l'usage d'un epsilon ou modifier la forme de certaines équations, lire (toute la série si vous le pouvez) https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

Unit in the Last Place (ULP)

- Une **ULP** -*aussi appelée Unit of Least Precision*- est la plus petite valeur représentable telle que l'addition de ULP et 1.0 donne une valeur différente de 1.0.
  - (Ceci revient à ajouter 1 à la représentation binaire du float).
  - On parle aussi parfois de « machine epsilon » cf https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

- **FLT_EPSILON** de la lib C standard représente un ULP.
- **FLT_MIN** représente quant à elle la plus petite valeur **normalisée** représentable en IEEE754

- La différence **absolue** abs(a-b) entre deux float de même signe, non identiques et différents de zéro nous donne une mesure de la précision avec une unité claire.
- La différence **relative** c=a-b, abs(c-0.f) n'est pas une mesure correcte de la précision car elle se compare à zéro. Et on sait que les float ont une précision accrue près de zéro.
  - Pour que la différence relative soit pertinente il faudrait comparer la valeur absolue abs(a-b) avec le plus grand des deux float multiplié par la précision acceptable, comme FLT_EPSILON par exemple

Exploiter FLT_MIN

- Il est alors possible d'exploiter ce savoir dans le cadre de l'optimisation de certaines fonctions.

- Dans le cas de la fonction réciproque 1/x on sait que x doit être différent de zéro, ce qui peut arriver par exemple dans un calcul de normalisation si la longueur d'un vecteur vaut zéro.

- En ajoutant à x une valeur plus petite qu'un ULP telle FLT_MIN=$1.0e^{-037}$ on modifie 0.0 mais de manière non significative de sorte que la précision et surtout le résultat du calcul ne sont pas altérés.
  - Le résultat est alors $1/(0 + 1.0e^{-037})$ ce qui produit une valeur différente de NaN et ne provoque pas d'exception.

- L'intérêt est ici d'éviter d'ajouter un test à chaque fonction pour s'assurer que la valeur est non nulle. Dans notre cas de figure on obtiendrait toujours le vecteur (0,0,0) pour une longueur de 0.

Même concept en SSE

- Prenons cette fois le cas d'une normalisation de vecteurs.
- Après avoir calculé la somme au carré (produit scalaire du vecteur avec lui-même) de 4 vecteurs on souhaite appliquer la fonction **_mm_rsqrt_ps**.

- Cependant, il est possible que l'une de ces 4 valeurs soit égale à 0 ce qui produirait NaN.

- Plutôt que de faire 4 tests et des branchements pour éviter la division par une racine à 0, il est possible de faire 1 seul test générant un masque qui vaut 0xffffffff pour les valeurs passant le test, et 0x0000000 dans le cas contraire.

```
// vec contient x²+y²+z²+w² pour 4 vecteurs traités en parallèle
__m128 mask = _mm_cmpgt_ps(vec, zero);
__m128 invSqrt = _mm_rsqrt_ps(vec);
__m128 res = _mm_and_ps(vec, invSqrt);
```

Architecture CPU avancée

Quelques définitions

- **µ-op**s : abréviation de micro-opérations. les processeurs *out-of-order* peuvent découper l'exécution d'une instruction en plusieurs parties appelées micro-opérations.

- Les processeurs modernes qui ont un héritage CISC (comme x86) décodent les instructions de façon RISC (µ-ops précablées) avec un *fallback* CISC pour les instructions microcodées ou exécutées rarement.

- **unité d'exécution** : un processeur dispose de plusieurs unités d'exécutions généralement par catégories de µops. Une unité d'exécution ne peut exécuter qu'une µ-op à la fois.
  - Il est donc important de savoir quelles sont les unités d'exécution utilisées par une instruction afin de réduire en blocage (*stall*) et améliorer le parallélisme.
  - la notion de « ***execution port*** » spécifie qu'elle partie du CPU émet une µ-op (une à la fois !)

Réordonnancement des instructions

- Beaucoup de CPU modernes ont une exécution dite « ***out-of-order*** », dans le désordre.

- Cela désigne la capacité d'un CPU à modifier l'ordre séquentiel d'un programme.

- Afin de pouvoir réordonnancer les instructions le CPU doit donc être capable de détecter les dépendances. Elles sont de 3 types, les *read* étant toujours indépendantes des autres *read*:

  - **Read-after-Write** (RAW): le résultat (mémoire ou registre) est lu après une instruction de modification
  - **Write-after-Read** (WAR): la même adresse ou registre est modifié après une instruction de lecture
  - **Write-after-Write** (WAW): le résultat (adresse ou registre) est modifié après une instruction de modification

- Certaines architectures permettent également le « renommage » des registres afin de réduire les dépendances explicites. Ceci n'est possible que dans les cas WAR et WAW.
- Le cas RAW est appelé « data-dependance » et une telle instruction ne peut être réordonnancée.

Super-Pipeline

- Dans une architecture super-scalaire le processeur peut émettre (*issue*) plusieurs instructions en parallèle.
- le nombre d'étages (stages) du pipeline est cependant fixe.
  - La longueur (*width*) du pipeline désigne le nombre d'étages consécutifs
  - Cela correspond en pratique aux « *execution ports* », seule une µ-op peut s'exécuter sur un port
- mais, dans le but d'exécuter plusieurs instructions en parallèle, les étages sont répliqués ce qui créé un « super-pipeline ».
  - On parle alors de « profondeur (*depth*) du pipeline »
  - Lorsque deux instructions consécutives utilisent des « exécution port » différents le CPU émet (*co-issue*) les deux instructions en même temps (totalement, ou partiellement si certaines µ-ops partagent le même port).
- Dans tous les cas la vitesse du pipeline dépend de l'étage le plus lent
- Le nombre total d'étapes dépend de l'architecture (ex: Pentium IV = 31 étages, Haswell = 14 à 19 étages selon la version du CPU…).

Read-Modify-Write / Load-Hit-Store

- Certaines des unités d'exécution des micro-opérations sont dédiées aux lectures et écritures
  - la phase d'écriture est splittée en deux, store (caches) et write-back (ram)

- Mais cela peut aussi conduire à des interblocages dus à la présence de dépendances ou aux accès mémoire (présence en cache, etc…)
  - On désigne souvent ces blocages sous le terme LHS (load-hit-store)

- Dans le cas d'une dépendance *Read-after-Write (RAW)* le pipeline sera bloqué à l'étape de « fetch » tant que l'instruction précédente n'aura pas fini l'étape « store » voire « write-back ».

- Plus généralement on parle de « ***pipeline stall*** », blocage du pipeline, lorsque le pipeline se retrouve bloqué à une étape.

Réduire les pipeline stalls

- Le réordonnancement des instructions (automatique ou manuel) est souvent une solution efficace.
- Dans le cas des data-dependance (RAW) il faut veiller à ce que le résultat ne soit pas différent.

- Exemple en pseudo code : on ajoute le contenu de R2 aux éléments d'un tableau pointé par R1. R3 contient l'adresse de fin du tableau.

```
Loop:                   ; 8 cycles par data
    LOAD R0, 0[R1]      ; 1 cycle
    ADD R0, R2          ; 1 cycle mais RAW entre R0 et R2 => stall de 1 cycle
    STORE 0[R1], R0     ; 1 cycle mais RAW entre R0 et la mémoire => stall de 2 cycles
    ADD R1, 8           ; 1 cycle car RAW de TEST
    TEST R1, R3         ; 1 cycle car RAW sur R1
    JNE Loop
```

Réduire les pipeline stalls : réordonnancement d'instruction

- On remarque que la première dépendance implique un stall d'1 cycle.
- En insérant une instruction d'une même durée et non dépendante on peut alors réduire la boucle d'un cycle

```
Loop:                   ; 7 cycles par data
    LOAD R0, 0[R1]      ; 1 cycle
    ADD R1, 8           ; 1 cycle, la dépendance ne bloque que l'écriture de R1 pas de cycle supp.
    ADD R0, R2          ; 1 cycle
    STORE -8[R1], R0    ; 1 cycles mais RAW entre ADD R0 et TEST R1 => stall de 2 cycles
    TEST R1, R3         ; 1 cycle
    JNE Loop
```

- La seule façon de réduire les stalls entre ADD et STORE est d'éffectuer des opérations non dépendantes mais ce n'est pas toujours possible.

Réduire les pipeline stalls : Dérouler la boucle

```
Loop:                   ; 14 cycles toutes les 4 data = 3.5 cycles par data
    LOAD R0, 0[R1]      ; 1 cycle
    LOAD R4, 8[R1]      ; 1 cycle
    LOAD R5, 16[R1]     ; 1 cycle
    LOAD R6, 24[R1]     ; 1 cycle
    ADD R0, R2          ; 1 cycle
    ADD R4, R2          ; 1 cycle
    ADD R5, R2          ; 1 cycle
    ADD R6, R2          ; 1 cycle
    STORE 0[R1], R0     ; 1 cycle
    STORE 8[R1], R4     ; 1 cycle
    ADD R1, 32          ; 1 cycle
    STORE -16[R1], R5   ; 1 cycle
    STORE -8[R1], R6    ; 1 cycle
    TEST R1, R3         ; 1 cycle
    JNE Loop
```

Very Large Instruction Word (VLIW)

- Un processeur VLIW est caractérisé par un pipeline très large mais peu profond car les instructions sont très longues afin de pouvoir combiner plusieurs instructions non dépendantes.
  - A ce titre les processeurs vectoriels comme le SSE/AVX représentent une forme de VLIW.

- Le compilateur doit donc effectuer un travail de *batching* des instructions de sorte à remplir le mieux possible l'opcode d'une instruction VLIW.
- Cependant le compilateur est limité par le manque de contexte (par exemple dans le cas des branchements, il ne peut utiliser les techniques de prédictions de branchement du CPU …).
  - Le compilateur devient plus complexe notamment lorsqu'il doit générer du code pour des architectures VLIW et non VLIW.

- Les Core i7 ont notamment un pipeline très large de type VLIW
  - de l'ordre de 14 étapes et n'ont une profondeur que de l'ordre de 6 opérations.

Dynamic scheduling

- Le CPU est également capable de réordonnancer le flux d'exécution des instructions de façon à réduire les blocages.

- C'est d'une aide importante car:
  - Toutes les dépendances ne sont pas connues au moment de la compilation
  - Les variations d'architecture d'une génération (voire au sein d'une même génération) de processeur sont importantes,
    - difficile de générer un code optimal pour tous les cas

- L'historique de prédiction de branchement est fondamental dans le cas de l'évaluation des branchements.

- Le renommage des registres est sans doute la technique la plus utilisée.

SIMD – en pratique

Adresses mémoires non alignées

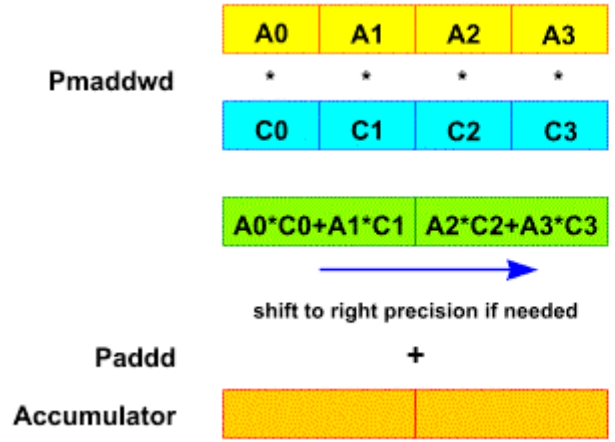
- Dans le cas où l'adresse mémoire n'est pas multiple de la taille d'un registre (8 octets en MMX, 16 octets en SSE, 32 en AVX …) il faut utiliser une instruction spéciale autrement une exception est levée.

- **_mm_loadu_***() / **_mm_storeu_***() et les équivalents AVX _mm256_ permettent de lire tout type d'adresse.
  - En assembleur cela correspond à l'instruction movdqu dans les deux cas
    - Historiquement lldqu était plus optimale car elle combinait les lignes de caches mais depuis les architectures core2 c'est le cas de movdqu
    - https://software.intel.com/en-us/forums/intel-isa-extensions/topic/777534

Exemple MMX – Produit scalaire (short int)

- La fonction **_mm_madd_pi16**() -équivalent de **_m_pmaddwd**()- effectue 3 opérations :
  - **multiplication** des opérandes signés 4*16 bits, et **extension** à 32 bits
  - **Addition** des 4 résultats, deux à deux, parties hautes et parties basses
- La fonction **_mm_srli_si64**() décale de N bits vers la droite (ici 32)
  - On stocke ici le résultat dans un autre registre
- La fonction **_mm_add_pi32**() –équivalent de **_m_paddd**()- effectue la somme parallèle de deux registres.

- Note:
  - fonctionne sur une représentation à virgule fixe sur 16 bits



Le produit scalaire en SSE

- On applique le même principe vue pour le MMX mais sur 4 floats

```
// (x0∗x1, y0∗y1, z0∗z1, w0∗w1)
__m128 t0 = _mm_mul_ps(v0, v);
// (y0∗y1, x0∗x1, w0∗w1, z0∗z1)
__m128 t1 = _mm_shuffle_ps( t0, t0, _MM_SHUFFLE(2 , 3 , 0 , 1));          // 0xB1
// (x0∗x1 + y0∗y1, x0∗x1 + y0∗y1, z0∗z1 + w0∗w1, z0∗z1 + w0∗w1)
__m128 t2 = _mm_add_ps(t0 , t1);
// ( z0∗z1 + w0∗w1, z0∗z1 + w0∗w1, x0∗x1 + y0∗y1, x0∗x1 + y0∗y1)
__m128 t3 = _mm_shuffle_ps(t2, t2, _MM_SHUFFLE(0 , 0 , 2 , 2));          // 0x0A
// ( dot , dot , dot , dot )
__m128 dotSplat = _mm_add_ss(t2, t3);
```

Le produit scalaire en SSE

- une autre instruction du SSE est intéressante, il s'agit de **_mm_movhl_ps**(__m128 a, __m128 b).
- Le résultat contient les éléments hauts de 'b' dans ses éléments bas, et les éléments hauts de 'a' dans ses éléments hauts. Théoriquement plus rapide qu'un _mm_shuffle_ps() sur toutes les architectures

```
// (x0∗x1, y0∗y1, z0∗z1, w0∗w1)
__m128 t0 = _mm_mul_ps(v0, v);
// (y0∗y1, x0∗x1, w0∗w1, z0∗z1)
__m128 t1 = _mm_shuffle_ps( t0, t0, _MM_SHUFFLE(2 , 3 , 0 , 1));    // 0xB1
// (x0∗x1 + y0∗y1, x0∗x1 + y0∗y1, z0∗z1 + w0∗w1, z0∗z1 + w0∗w1)
__m128 t2 = _mm_add_ps(t0 , t1);
// ( z0∗z1 + w0∗w1, z0∗z1 + w0∗w1, x0∗x1 + y0∗y1, x0∗x1 + y0∗y1)
__m128 t3 = _mm_movhl_ps(t1, t2);
// ( dot , dot , dot , dot )
__m128 dotSplat = _mm_add_ss(t2, t3);
```

Le produit scalaire en SSE3

- La fonction **_mm_hadd_ps**() réalise la somme séparée des 2 parties hautes et 2 partie basses et stocke le résultat dans les 2 parties basses
- Il faut donc l'utiliser deux fois d'affilée avec les mêmes registres afin de produire une somme horizontale complète.

- Sur les architectures modernes, la latence est de 5 à 6 cycles suivant la génération de processeur, et le *reciprocal throughput* est de 2, au minimum.
  - Ce qui nous donne au minimum une latence de 10/12 cycles car nous avons ici une dépendance entre les deux appels de _mm_hadd_ps().

- Ceci est accentué par le fait que la prochaine instruction similaire ne pourrait s'exécuter que 2 cycles après la fin de précédente du fait du throughput de 2.
  - On est donc sur une latence minimale de 12 à 14 cycles pour cette alternative.
  - Instruction à réserver pour des cas de figure où l'on a beaucoup de données à traiter

Le produit scalaire en SSE4

- **_mm_dp_ps**()

- Latence de 13 en moyenne, et un *reciprocal throughput* de 1.4 à 2.0.

- Comme pour le cas précédent, c'est un cas typique où une instruction simple à utiliser est par contre plus lente que ce qui existait jusqu'à présent.

- Ces instructions sont généralement prévues pour être utilisées avec un nombre de données importantes de sorte à ce que différentes instructions (et différents *execution ports*) puissent s'exécuter en parallèle.

Limites de la vectorisation

- Je vous invite à lire ce document qui argumente le point que je vais exposer http://www.farbrausch.de/~fg/articles/ubiquitous_sse_vector.html

- En substance, l'usage du SIMD n'a de sens qu'à partir du moment où le traitement s'effectue de sorte à maximiser le parallélisme et le *throughput*.
- Plus concrètement, utiliser des instructions SIMD pour n'effectuer qu'une banale opération réelle ou vectorielle n'apporte pas un gain intéressant vue l'effort fourni (sachant que de nos jours le SSE est déjà utilisé par défaut pour les opérations sur les float).
- De plus, les temps de load/store des valeurs de la mémoire vers les registres SIMD peuvent rendre le code plus lent qui prévu (encore une fois, le SSE est généré par défaut pour les float donc ce n'est pas aussi bloquant que cela de ns jours).
    - Notre produit scalaire est donc un contre-exemple assez évident.

- Par contre, si notre code est adapté pour effectuer le plus d'opérations possibles à l'instar des principes de la D.O.D (« when there's one there's many »), dans ce cas cela se révèle plus payant.

Le produit scalaire SIMD : 4 à la fois

- On l'a vu, un premier obstacle vient du fait que les composantes x, y, z, w ne sont pas additionnable sans instructions horizontales.

- En réarrangeant nos données de sortes à avoir un tableau de 4+ composantes x, 4+ composantes y etc… on peut tout à la fois simplifier le code

- Notre classe vec4 devient alors une structure de tableau (Structure Of Array, SOA) plutôt qu'un tableau de struct vec4 (Array Of Structures, AOS) dans une logique POO.
  - Ce n'est pas un changement évident et simple mais il est nécessaire dans notre cas.

- A noter que ce n'est pas tant la structure (AOS vs SOA) qui importe mais le principe DOD de répétition d'un traitement sur plusieurs données d'affilées.

Cas d’une Matrix4

- Pratiquement, il y’a une certaine quantité d’instructions à exécuter pour que le gain se fasse ressentir avec des coprocesseurs, ici en SIMD.
- Le gain sera maximisé à partir du moment où l’on effectue le même type de calcul en lot (batch) ce qui idéalise le fonctionnement du CPU
  - meilleure cohérence et utilisation des caches car les données sont adjacentes en mémoire, répétition d’un même code stocké dans le cache d’instruction…

- Si l’on prend le cas d’une multiplication Matrix4*vec4 ou Matrix4*Matrix4 on se retrouve dans des cas de figure un peu plus intéressants que vec4*vec4 (ou que le produit scalaire).
- On sait d’ailleurs qu’un produit matriciel est décomposable en produits scalaires. Cependant ce n’est pas la meilleure façon de procéder.
- On Remarque en effet que dans l’expression M*v les premiers éléments de chaque colonne sont multipliées avec v.x, la seconde ligne avec v.y etc…

Matrix4 * vec4

- On va ici opérer un *splatting* (réplication) des éléments du vecteur à droite de sorte à générer v.xxxx, v.yyyy, v.zzzz et v.wwww

```
__m128 vx = _mm_shuffle_ps(v, v, _MM_SHUFFLE(0,0,0,0));          // 0x00
__m128 vy = _mm_shuffle_ps(v, v, _MM_SHUFFLE(1,1,1,1));          // 0x55
__m128 vz = _mm_shuffle_ps(v, v, _MM_SHUFFLE(2,2,2,2));          // 0xAA
__m128 vw = _mm_shuffle_ps(v, v, _MM_SHUFFLE(3,3,3,3));          // 0xFF
__m128 t0 = _mm_mul_ps(M[0], vx);
__m128 t1 = _mm_mul_ps(M[1], vy);
__m128 t2 = _mm_mul_ps(M[2], vz);          // entrelacer les shuffle et mul peut être plus rapide dans certains cas
__m128 t3 = _mm_mul_ps(M[3], vw);
t0 = _mm_add_ps(t0, t1);                   // note : Instruction Level Parallelism
t2 = _mm_add_ps(t2, t3);                   // remarquez que le deux _mm_add_ps sont indépendants
__m128 res = _mm_add_ps(t0, t2);
```

A lire également https://fgiesen.wordpress.com/2015/02/05/a-small-note-on-simd-matrix-vector-multiplication/

Contrôle du flux

- Certains algorithmes nécessitent l'usage de tests (if / else) ce qui donne lieu à différentes combinaisons de flux d'instructions.
- C'est forcément l'aspect qui « casse » la possible vectorisation du code.

- On a déjà vu l'usage des masques tout au début dans le cas de sqrt.
- L'idée va donc consister à exploiter au maximum les résultats des comparaisons en parallèle afin de sélectionner le résultat.

- Comme on ne peut choisir une branche par rapport à une autre il est souvent nécessaire d'évaluer les deux branches à la fois. On fait donc en premier la comparaison (le test) puis on combine les résultats de l'évaluation des deux (ou plus) branches.

- En SSE4.1, **_mm_blendv_ps**() et ses variantes est pratique et plus rapide que les solutions classiques à base d'opérations logiques. Par exemple un selecteur ternaire (cond?a:b) se fait traditionnellement ainsi **_mm_or_ps(_mm_and_ps**(a, cond), **_mm_andnot_ps**(cond, b))
  - Où cond est le résultat d'un test **_mm_cmp**_ps**() par exemple

Contrôle du flux

- On peut également traiter les simples tests booléens comme des facteurs d'un calcul. En pseudo-code:

```
For each Particule P
    if P.isAlive
        P.position += P.velocity * dt
```

- Devient

```
For each Particule P
    P.position += P.velocity * dt * (P.isAlive&1)        // un booléen n'est pas forcément stocké sur un bit
```

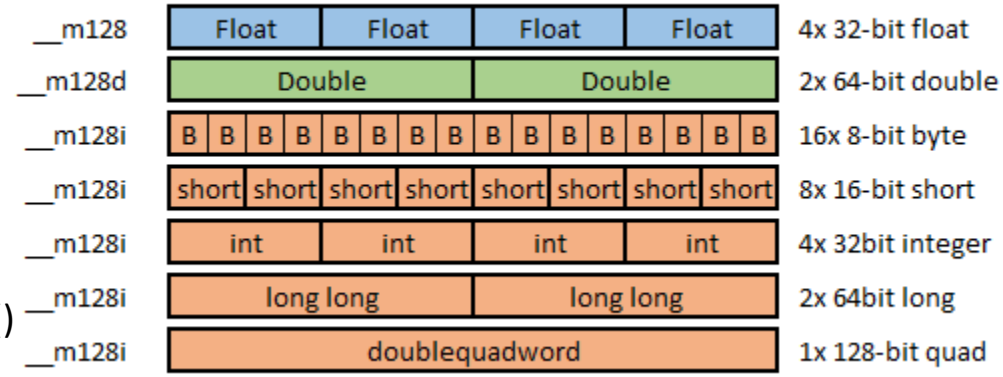
- Quatre à la fois

```
For each Particules P0->3
    For i=0->3
        P.position.x[i] += P.velocity.x[i] * dt * (P.isAlive[i]&1)
        …
```

SSE2 – instructions MMX étendues

- En plus du support des double-s, Le SSE2 étends le jeu d'instruction MMX à 128 bits.

- Le type **__m128i** a de multiple alias
- Le type **__m128d** est à utiliser pour les double
- Le type **__m64** est le type 64 bits entier du MMX
- Notez qu'il est différent de int64_t
  - On peut convertir d'un type vers l'autre à l'aide de **_m_from_int**() **_m_from_int64**(), **_mm_cvtm64_si64**() et **_mm_cvtsi64_m64**()



- La syntaxe des instructions SSE2 est assez similaire à celle du MMX (sauf que les registres MMX sont séparés)
  - A quelques exceptions près, les instructions MMX agissent toujours en packed
  - De même rares sont les instructions agissant sur des entiers non-signées '_pu'

- MMX **_mm_add_pi8**(__m64 a, __m64 b) // et _pi16, _pi32
- SSE2 **_mm_add_epi8**(__m128i a, __m128i b) // 'e' pour 'extended', et _epi16, _epi32, _epi64

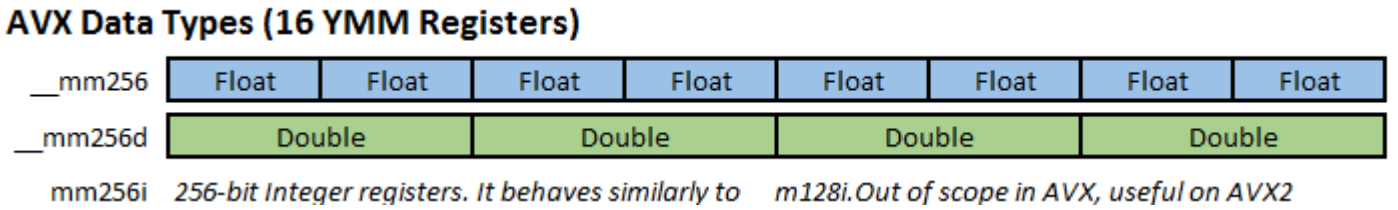
Conversion int <-> float

- En SSE **_mm_cvtss_si32**() effectue un arrondi tandis que **_mm_cvttss_si32**() effectue une troncation. Ces deux fonctions renvoient un ***int*** plutôt qu'un registre XMM.

- Toujours en SSE, __m128 **_mm_cvt_si32ss(__m128** a, **int** b) converti un int en second paramètre vers un __m128 en prenant les float de poids supérieur depuis le premier paramètre.

- En SSE2, **_mm_cvtsi32_ss**() est similaire à **_mm_cvt_si32ss**() en plus simplifié.

Transition vers AVX



- De la même manière que le code x86 peut fonctionner sur une architecture x86_64, le code SSE peut fonctionner sur un CPU disposant de registres et instructions AVX et AVX512 et leurs variantes…
- …cependant, il y'a quelques subtilités liées au fait que les registres ont une plus grande taille.

- La syntaxe **VEX** (Vector EXtension) est un nouvel opcode introduit avec les instructions AVX qui se caractérise par un préfixe ajouté aux instructions SSE (tel **v**movap**s**/**v**movap**d** en assembleur).
  - vmovaps est donc une instruction AVX (VEX) qui utilise les registres XMM (ps au lieu de pd)
  - Correspond à un _mm256_load/store_ d'un registre __m256 depuis/vers un registre __m128

- Les registres XMM sont partagés avec les registres YMM mais pour que l'AVX puisse être utilisé il est important de veiller à ce que les octets de poids forts YMM soient bien initialisés à zéro.
  - Cette initialisation fait partie de l'ABI, elle est donc réalisée automatiquement par les compilateurs (possible de désactiver cela pour plus de contrôle).
  - C'est particulièrement important lorsqu'une instruction AVX manipule un registre XMM.

- De plus cette règle prévaut également dans le cas de l'exécution multitâche. Le processeur doit obligatoirement préserver l'ensemble des registres YMM si il n'est pas sûr que la partie haute contient zéro.

Produit scalaire en AVX ?

- Comme on peut le remarquer, on a maintenant 8 float-s par registre ce qui ne colle pas avec un produit scalaire traditionnel.
- Trois possibilités :
- Se limiter au SSE
- Essayer de faire deux produits scalaires à la fois
  - C’est la solution la moins intéressante car elle requiert beaucoup de shuffle etc…
- Raisonner DOD en effectuant 8 produits scalaires en même temps

Transition vers AVX, éviter les pénalités

- Afin d'éviter que le processeur sauvegarde à tout va dès que notre code utilise des fonctions SSE il y'a plusieurs possibilités:

- Utiliser le préfixe **VEX** (**_mm256_** pour les intrinsèques) plutôt que _mm_.
  - A noter que les mnémoniques AVX utilisent 3 opérandes au lieu de 2.
    - Le compilateur devrait insérer des instructions pour la transition (cf plus bas)
    - Certaines (rares) instructions SSE n'ont pas d'équivalent VEX, telle **_mm_movpi64_epi64**()

- Insérer manuellement des instructions dédiées à l'initialisation à zéro des octets de poids fort telle _mm256_zeroupper() **après** les instructions AVX et **avant** les instructions SSE
  - Ainsi le CPU est informé de l'état complet du registre et n'a pas besoin de sauvegarder le contexte YMM.
    - _mm256_zeroall() marche également mais efface l'ensemble du registre
  - Notez bien que seules ces deux instructions permettent d'éviter les pénalités, un xor ou autre fonction produisant zéro n'invaliderait pas la transition.

Strlen version SIMD

- En SSE2 on peut reprendre le même principe que l'on a vu pour strlen version assembleur :

```
__m128i zero = _mm_setzero_si128();
for(;;) {
    __m128i chars = _mm_load_si128((__m128i)str);
    chars = _mm_cmpeq_epi8(chars, zero);
    if ((mask = _mm_movemask_epi8(chars) != 0) {
        _BitScanForward(&pos, mask); // recherche l'indice du bit à 1
        len += pos;
        break;
    }
    str += sizeof(__m128i); len += sizeof(__m128i);
}
```

- En SSE4.2 **_mm_cmpistri**(a, b, mode) remplace la comparaison et le test (avec mode = _SIDD_CMP_EQUAL_EACH)

Extension F16C

- Le format binary16 (float16) est supporté sur la plupart des processeurs actuels mais malheureusement seulement pour les conversions float16<->float32

- Cela reste cependant utile pour réduire l'empreinte mémoire des données lorsque 16 bits suffisent,
  - typiquement pour convertir à la volée des images/sons au format float32 vers float16
  - https://software.intel.com/en-us/articles/performance-benefits-of-half-precision-floats
  - https://software.intel.com/en-us/blogs/2013/09/30/intel-half-precision-floating-point-format-conversion-instructions

Extension FMA3 (Fused Multiply-Add)

- Lorsque les CPUs supportent le FMA, les compilateurs sont capables d'émettre des instructions de combinaisons (*fuse*) des multiplications et additions.
  - C'est le cas des processeurs AMD depuis de nombreuses années et Intel Haswell et supérieurs.

- Si l'extension FMA3 est supportée mais qu'on a un doute sur les capacités du compilateur à combiner, les fonctions intrinsèques **_mm_fmadd_ps/d**(), **_mm256_fmadd_ps/d**() etc… peuvent être utilisées.

Annexes

Méthode de Newton-Raphson appliquée au SSE

Méthode Newton-Raphson

- Les instructions SSE sont très rapides mais cela se fait parfois au détriment de la précision.
  - En interne elles utilisent une LUT de quelques dizaines de bits.
  - C'est le cas notable des fonctions _mm_rcp_*, _mm_sqrt_*, _mm_rsqrt_*.

- La méthode de Newton-Raphson est une technique itérative analytique qui va nous permettre d'augmenter la précision autant que souhaité.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

  - Le coût est relativement moyen, donc pas négligeable, mais rien qu'une seule itération permet d'augmenter la précision à 2/3 décimales supplémentaires.
  - Attention, pour que le résultat soit pertinent il faut que les données initiales soient déjà d'une précision acceptable. C'est le cas avec sqrt et rsqrt du SSE.

- Il se trouve par ailleurs que $\sqrt{x}$ et $1/\sqrt{x}$ sont des cas particuliers de cette méthode

Méthode Newton-Raphson : $\frac{1}{x}$

- La fonction rcp(x) correspond à $\frac{1}{x}$ ou x^{-1}.
- A partir de maintenant x va être le résultat de l'appel de rcp(a) : $x = \frac{1}{a}$
  - où a est le nombre dont on souhaite calculer l'inverse

- x est alors une première approximation de $\frac{1}{a}$, et il nous faut $f(x)$ afin d'appliquer la méthode
- On résout d'abord pour a, ce qui donne $\frac{1}{x} = a$, la racine étant alors $\frac{1}{x} - a = 0$ d'où

  $f(x) = \frac{1}{x} - a$ et $f'(x) = -\frac{1}{x^2}$
  - https://www.maths-france.fr/Terminale/TerminaleS/FichesCours/FormulesDerivees.pdf

- la formule récurrente est y = $x - {}^{f(x)}/_{f'(x)}$

$$= x - \frac{\frac{1}{x} - a}{-\frac{1}{x^2}} = x - \frac{\frac{1-ax}{x}}{-\frac{1}{x^2}} = x - \frac{-(1-ax)x^2}{x} = x + (1-ax)x = x + x - ax^2 = 2x - ax^2 = \boldsymbol{x(2-ax)}$$

Méthode Newton-Raphson appliquée à rcp

- On part de

__m128 x = **_mm_rcp_ss**(a);

- et on doit appliquer $y = x(2 - ax)$

__m128 two = **_mm_set1_ps**(2.f);
__m128 a_x = **_mm_mul_ss**(a, x);
__m128 y = **_mm_mul_ss(_mm_sub_ss**(two, a_x), x);

Méthode Newton-Raphson : $\sqrt{x}$

- La fonction sqrt(x) correspond à $\sqrt{x}$ ou $x^{0.5}$.
- En s'inspirant de la formule de la méthode de Héron on pose $x = \sqrt{a}$
  - où a est le nombre dont on souhaite calculer la racine

- x est une première approximation de $\sqrt{a}$, et il nous faut $f(x)$
- En mettant au carré on a $x^2 = a$, la racine étant alors $x^2 - a = 0$ d'où

  $f(x) = x^2 - a$ et $f'(x) = 2x$
  - https://www.maths-france.fr/Terminale/TerminaleS/FichesCours/FormulesDerivees.pdf

- la formule récurrente est y = $x - {}^{f(x)}/_{f'(x)}$

$$= x - \frac{x^2-a}{2x} = \frac{2x*x}{2x} - \frac{x^2-a}{2x} = \frac{x^2+a}{2x} = \frac{\boldsymbol{x+\frac{a}{x}}}{\mathbf{2}}$$

Méthode Newton-Raphson appliquée à sqrt

- On part de

__m128 x = **_mm_sqrt_ss**(a);

- et on doit appliquer $y = \frac{x+\frac{a}{x}}{2}$

__m128 half = **_mm_set1_ps**(0.5f);
__m128 a_over_x = **_mm_mul_ss**(a, **_mm_rcp_ss**(x));
__m128 y = **_mm_mul_ss(_mm_add_ss**(x, a_over_x), half);

Méthode Newton-Raphson : $\frac{1}{\sqrt{x}}$

- La fonction rsqrt(x) équivaut à $\frac{1}{\sqrt{x}}$, autrement dit $x^{-0.5}$.
  - $x^{0.5}$ correspond à $\sqrt{x}$, x^{-1} correspond à $\frac{1}{x}$, donc $x^{-0.5}$ correspond à $\frac{1}{\sqrt{x}}$.
- En s'inspirant de la formule que nous avons vu pour $\sqrt{x}$ on pose $x = \frac{1}{\sqrt{a}}$
  - où a est le nombre dont on souhaite calculer la racine

- x est une première approximation de $\frac{1}{\sqrt{a}}$, et il nous faut $f(x)$
- En inversant puis en mettant au carré on résout l'équation pour a: $\frac{1}{x^2} = a$,
- alors $f(x) = \frac{1}{x^2} - a = 0$ et $f'(x) = -\frac{2}{x^3}$
- la formule récurrente est toujours y $= x - {}^{f(x)}/_{f'(x)}$

$$= x - \frac{\frac{1}{x^2} - a}{-\frac{2}{x^3}} = x\,\frac{-\frac{2}{x^3}}{-\frac{2}{x^3}} - \frac{\frac{1}{x^2} - a}{-\frac{2}{x^3}} = \frac{-\frac{2x}{x^3} - \frac{1}{x^2} + a}{-\frac{2}{x^3}} = \frac{-\frac{3}{x^2} + a}{-\frac{2}{x^3}} = \frac{-\frac{3+ax^2}{x^2}}{-\frac{2}{x^3}} = \frac{(-3+ax^2)x^3}{-2x^2} = \frac{(3-ax^2)x}{2} = \boldsymbol{x\left(\frac{3}{2} - \frac{a}{2}x^2\right)}$$

Méthode Newton-Raphson appliquée à rsqrt

- On part de

__m128 x = **_mm_rsqrt_ss**(a);

- et on doit appliquer $y = \frac{(3-ax^2)x}{2}$

__m128 half = **_mm_set1_ps**(0.5f), three = **_mm_set1_ps**(3.f);
__m128 a_mul_x2 = **_mm_mul_ss**(a, **_mm_mul_ss**(x, x));
__m128 y = **_mm_mul_ss(_mm_s_sub**(three, a_mul_x2), half);

Références

- https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- https://software.intel.com/sites/landingpage/IntrinsicsGuide
- https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf
- https://www.naic.edu/~phil/software/intel/11MC12_Avoiding_2BAVX-SSE_2BTransition_2BPenalties_2Brh_2Bfinal.pdf
- http://www.farbrausch.de/~fg/articles/ubiquitous_sse_vector.html
- http://arith22.gforge.inria.fr/slides/s1-cornea.pdf