Évolution des architectures et des jeux d'instructions SIMD sur les processeurs généralistes

Sorbonne Université – Master SESI – MU5IN166 – Hot Topics

Adrien Cassagne

Le 12 décembre 2023

Table des matières

Qui suis-je ?

1 Introduction

Adrien Cassagne, enseignant-chercheur (MCF) à Sorbonne Université (LIP6)

- Fort intérêt pour les **architectures programmables généralistes**
- **Optimisation de code** pour des cibles spécifiques (CPU / GPU)
- Implémentation efficace d'algorithmes (en **temps** et en **énergie**)
  - Besoin de connaître les **architectures matérielles**
  - Besoin de **comprendre les algorithmes**

Qui suis-je ?

1 Introduction

Adrien Cassagne, enseignant-chercheur (MCF) à Sorbonne Université (LIP6)

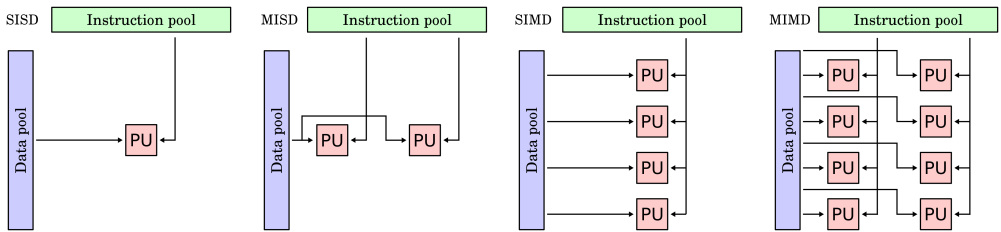
- Fort intérêt pour les **architectures programmables généralistes**
- **Optimisation de code** pour des cibles spécifiques (CPU / GPU)
- Implémentation efficace d'algorithmes (en **temps** et en **énergie**)
  - Besoin de connaître les **architectures matérielles**
  - Besoin de **comprendre les algorithmes**

Quelques domaines d'application

- Communications numériques (codes correcteurs d'erreurs)
- Traitement de l'image et de la vidéo (détection du mouvement, météores)
- Dynamique des fluides (calcul de l'écoulement des fluides, aéronautique)

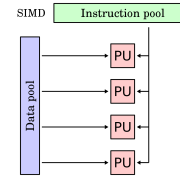
Taxonomie de Flynn

Une classification des architectures par Michael Flynn en 1966.



Le modèle *Single Instruction Multiple Data* (SIMD)

1 Introduction



- Adapté pour du **calcul régulier**
- Permet d'augmenter la puissance de calcul de l'architecture
- Traitement sur **des vecteurs d'éléments**

```
1 void vadd (float A[N],
2            float B[N],
3            float C[N]) {
4   for (int i = 0; i < N; i++) {
5     C[i] = A[i] + B[i];
6   }
7 }
```

```
1 void vadd (float A[N],
2            float B[N],
3            float C[N]) {
4   for (int i = 0; i < N; i+=4) {
5     C[i+0] = A[i+0] + B[i+0];
6     C[i+1] = A[i+1] + B[i+1];
7     C[i+2] = A[i+2] + B[i+2];
8     C[i+3] = A[i+3] + B[i+3];
9   }
10 }
```

Si on peut faire $C[i:i+4] = A[i:i+4] + B[i:i+4]$ en **une seule instruction** alors on peut aller potentiellement 4 fois plus vite !

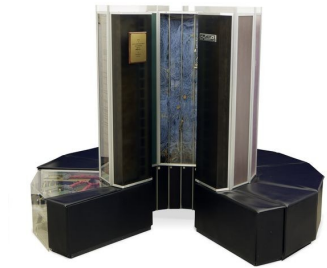
Table des matières

2 À l'origine, il y avait les machines vectorielles puis vint le SIMD

Origine des architectures SIMD : les supercalculateurs vectoriels

2 À l'origine, il y avait les machines vectorielles puis vint le SIMD

- La machine ILLIAC IV de l'Université de l'Illinois en 1972 (100 $\simeq$ 150 MFlops)
- Le supercalculateur Cray-1 en 1976 (160 MFlops)



Les supercalculateurs sont utilisés pour effectuer les calculs (souvent scientifiques) les plus gourmands. Ils sont à la pointe de la technologie et bien souvent précurseur sur les architectures grand public.

Les supercalculateurs vectoriels

2 À l'origine, il y avait les machines vectorielles puis vint le 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 jumplt i, 10000, loop # jump if i < 10000
```

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 `c` est **calculé en séquentiel** dans les deux cas

Single Instruction Multiple Data (SIMD)

2 À l'origine, il y avait les machines vectorielles puis vint le 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 # jump if i < 10000
```

Exemple d'instructions **SIMD** :

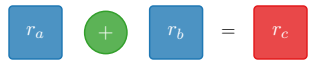
```
1  loop:
2    loadv va, [p_a] # lecture de 4 éléments
3    loadv vb, [p_b] # lecture de 4 éléments
4    addv vc, va, vb # addition sur 4 éléments
5    storev [p_c], vc # écriture de 4 éléments
6    add p_a, p_a, 4*4 # 4*4 bytes, 128-bit
7    add p_b, p_b, 4*4 # 4*4 bytes, 128-bit
8    add p_c, p_c, 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 `vc` est **calculé en parallèle**

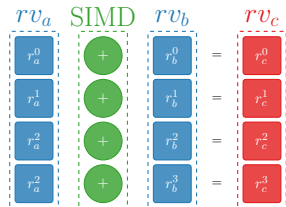
Single Instruction Multiple Data (SIMD)

2 À l'origine, il y avait les machines vectorielles puis vint le 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

2 À l'origine, il y avait les machines vectorielles puis vint le 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)

Table des matières

3 Père Castor, raconte-nous une histoire : les ordinateurs SIMD

L'arrivée des architectures SIMD "grand public": MMX

L'extension MMX (MultiMedia eXtension) d'Intel (x86) en 1997

- 8 registres 64-bit, MM0-MM7
- **Dédié au calcul d'entiers** (pas de flottants)
- $8 \times$ 8-bit, $4 \times$ 16-bit et $2 \times$ 32-bit

Pourquoi faire ?

- Du calcul 2D/3D (jeux vidéo entre autres)
- Traitement du signal (codec audio, vidéo)

Des problèmes ?

- Fonctionne exclusivement **sur des entiers**
- **Bloque l'utilisation des calculs flottants**
- Utilisation compliquée : code assembleur

L'arrivée des architectures SIMD "grand public": Altivec

L'extension Altivec pour PowerPC en 1999

- 32 registres 128-bit
- **Calcul d'entiers et de flottants** (simple précision)
- 16 × 8-bit, 8 × 16-bit et 4 × 32-bit
- Des instructions de "prefetch"
- Beaucoup plus riche que MMX !

Des limitations ?

- Pas de calcul 64-bit
- **Impossible de passer de registres scalaires à vectoriels**
- Contraintes d'alignement fortes

Streaming SIMD Extensions (SSE)

Pendant ce temps là, chez les bleus on prépare la contre offensive...

L'extension "*Streaming SIMD Extensions*"(SSE) sur x86

- 8 registres 128-bit, XMM0-XMM7
- **Calcul de flottants** (simple précision)
- 4×32-bit
- Les premier processeur est le Pentium III en 1999

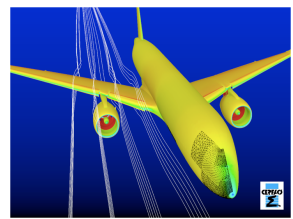
Des limitations ?

- Pas de calcul 64-bit, ni sur les entiers
- Contraintes d'alignement
- Pas encore aussi complet que Altivec

Streaming SIMD Extensions 2 (SSE2)

La version 2 de SSE sur x86 à partir du Pentium 4 en 2000

- 8 registres 128-bit en x86
- 16 registres 128-bit en x86-64 (2003, Athlon 64)
- **Calcul d'entiers** pour remplacer MMX
- **Calcul 64-bit** (entiers et flottants)
- 16 × 8-bit, 8 × 16-bit, 4 × 32-bit et 2 × 64-bit



Des limitations ?

- Contraintes d'alignement

Streaming SIMD Extensions 3 (SSE3)

Jamais 2 sans 3... La version 3 de SSE sur x86-64

- Des **opérations** dîtes "**horizontales**"
  - Exemple: HADDPS (Horizontal-Add-Packed-Single):
  - Entrées : $\{A_0, A_1, A_2, A_3\}, \{B_0, B_1, B_2, B_3\}$
  - Sortie :$\{A_0 + A_1, A_2 + A_3, B_0 + B_1, B_2 + B_3\}$
- Les chargements non-alignés sont moins pénalisants
- Pentium 4 (Prescott) en 2004 (ça chauffe...)

Pourquoi faire ?

Streaming SIMD Extensions 3 (SSE3)

Jamais 2 sans 3... La version 3 de SSE sur x86-64

- Des **opérations** dîtes "**horizontales**"
  - Exemple: HADDPS (Horizontal-Add-Packed-Single):
  - Entrées : $\{A_0, A_1, A_2, A_3\}, \{B_0, B_1, B_2, B_3\}$
  - Sortie :$\{A_0 + A_1, A_2 + A_3, B_0 + B_1, B_2 + B_3\}$
- Les chargements non-alignés sont moins pénalisants
- Pentium 4 (Prescott) en 2004 (ça chauffe...)

Pourquoi faire ?

- Parfois on a besoin de faire des calculs au sein d'un même registre vectoriel (3D, traitement du signal)
  - Solution 1 : on **réagence les données dans les registres** puis on calcule verticalement
  - Solution 2 : **le matériel prévoit certains réagencements** pour nous (SSE3)

Streaming SIMD Extensions 4 (SSE4)

Cela ne s'arrêtera donc jamais ?!

- Un ensemble d'instructions assez spécifique à certains domaines :
  - Calcul pour des nombres complexes en *Array of Structures* (AoS)
  - CRC 32-bit pour le polynôme `0x11EDC6F41`
  - Etc.
- Une instructions généraliste très utilisée : `blend`
- Core 2 Duo en 2008

```
int m = 1;
int a = 10;
int b = 20;
int c;
c = mask ? a + b : a - b;
// c = 30
```

```
int_x4 m = {1, 0, 1, 1};
int_x4 a = {10, 10, 10, 10};
int_x4 b = {20, 20, 20, 20};
int_x4 c;
c_p = instr_SIMD_add(a, b);
c_m = instr_SIMD_sub(a, b);
c = instr_SIMD_blend(m, c_p, c_m);
// c = {30, -10, 30, 30}
```

Advanced Vector Extensions (AVX)

Un nouveau jeu d'instruction en 2011 (x86 Sandy Bridge) :

- 16 registres 256-bit (registres **deux fois plus grand que SSE**)
- Uniquement des opérations sur **les nombres flottants**
- Pas beaucoup de différences en terme de fonctionnalités avec SSE
- 8×32-bit et 4×64-bit

À fréquence égale, la puissance de calcul crête est doublée par rapport à SSE !

Advanced Vector Extensions 2 (AVX2) et FMA

Version 2 d'AVX en 2013 (x86 Haswell) :

- Ajout du support des **nombres entiers**
- $32 \times$ 8-bit, $16 \times$ 16-bit, $8 \times$ 32-bit et $4 \times$ 64-bit
  - — Cela commence à faire beaucoup de parallélisme !!

Extension Fused Multiply and Add (FMA) : $d = a \times b + c$

- Le processeur est maintenant capable de **faire une multiplication et une addition en 1 cycle d'horloge**

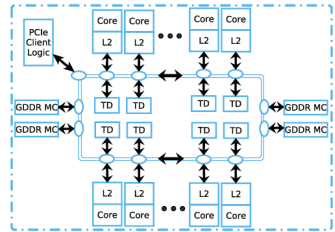
À fréquence égale, la puissance de calcul crête est doublée par rapport à AVX1 !

La tendance “manycore”

3 Père Castor, raconte-nous une histoire : les ordinateurs SIMD

Le principe : multiplier les cœurs “idiots”.

- Fréquence relativement basse
- Pas de caches ou des caches simples et pas très grands
- Faible *out-of-order* voire *in-order*
- Des ressemblances avec les GPUs
- Très large ISA SIMD : 512-bit (en 2012)



AVX-512

3 Père Castor, raconte-nous une histoire : les ordinateurs SIMD

Lancé en 2016 sur les Xeon Phi

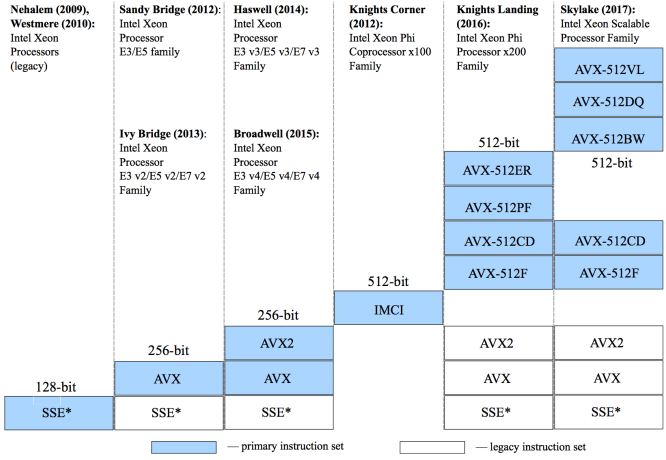
- Plus qu'une simple rallonge de vecteurs
- Des **instructions systématiquement masquées**
- Des **nouveaux registres de type masque**
- $64 \times$ 8-bit, $32 \times$ 16-bit, $16 \times$ 32-bit et $8 \times$ 64-bit

Mais à quoi ça sert ?

- Faire des chargement et de stockage sans *segfault*
- Avec et sans masquage = même coût en nombre de cycles (évite le surcoût des `blend` par ex.)
- **Augmente la complexité du jeu d'instructions**

Évolution des instructions SIMD sur x86

3 Père Castor, raconte-nous une histoire : les ordinateurs SIMD



Les mots bienveillants de Linus Torvalds (2020)

“J’espère que l’AVX-512 mourra d’une mort douloureuse, et qu’Intel commencera à résoudre les vrais problèmes au lieu d’essayer de créer des instructions magiques pour ensuite créer des benchmarks sur lesquels ils pourront s’appuyer.”

→ Les processeurs AMD Threadripper (Zen 2/3) sont environ 3 fois plus rapides pour compiler le noyau Linux...

Les mots bienveillants de Linus Torvalds (2020)

"J'espère que l'AVX-512 mourra d'une mort douloureuse, et qu'Intel commencera à résoudre les vrais problèmes au lieu d'essayer de créer des instructions magiques pour ensuite créer des benchmarks sur lesquels ils pourront s'appuyer."

→ Les processeurs AMD Threadripper (Zen 2/3) sont environ 3 fois plus rapides pour compiler le noyau Linux...

- Intel Alder Lake (nov. 2021) : suppression du support des instructions AVX-512
- AMD Zen 4 (sept. 2022) : ajout du support des instructions AVX-512 mais unités de calcul sur 256 bits

Des problèmes avec le SIMD ?

- Le **compilateur** n'est souvent **pas capable d'auto-vectoriser** correctement
- Comme pour la programmation multi-threads, c'est au programmeur d'écrire le code vectoriel (pas de magie)
- Bien souvent, **vectoriser** un code = **repenser l'algorithme** (point commun avec les GPUs)
- Un zoo d'ISA SIMD (pour faire court MMX, SSE, AVX, AVX-512), **nuit grandement à la portabilité du code**

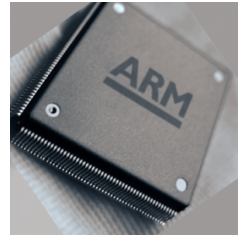
Table des matières

4 Le SIMD dans les systèmes embarqués

Processeurs ARM

4 Le SIMD dans les systèmes embarqués

- ARMv7 : NEON 1 128-bit
  - 2005
  - Très proche de SSE4.2
  - Pas de support des nombres 64-bit
- ARMv8 : NEON 2 128-bit
  - 2012
  - Très proche de SSE4.2
  - Support des nombres 64-bit
- Armv8.2-A : SVE1
  - 2017
  - 1er jeu d'instruction agnostique
- ARMv9 : SVE2
  - 2021
  - Très proche de SVE1



Processeurs ARM : SVE

ARM Scalable Vector Extension (SVE)

- Le binaire ne contient pas la taille des registres vectoriels
  - Ressemble aux “vieux” supercalculateur vectoriels
  - Objectif: **avoir des codes plus durable dans le temps**
- Première fois introduit dans le supercalculateur Fugaku
  - Implémenté avec des **unités SIMD de 512-bit**
  - **Supercalculateur le plus puissant** au monde en 2021
- Comme AVX-512, des **instructions masquées**

Processeurs RISC-V

RISC-V a le vent en poupe ces derniers temps

- **Un jeu d'instruction ouvert et modulable** (à base d'extensions)
- Des implémentations libres (sur FPGA ou pour ASIC)
- Une chaîne complètement libre et open source du matériel au logiciel !
- Très inspiré de MIPS et ARM

Un bon cheval de bataille pour la **souveraineté européenne dans les semi-conducteurs**...

Processeurs RISC-V : Extension Vectorielle

Une extension spécifique en **cours de normalisation** pour le **calcul vectoriel**

- RISC-V Vector extension (RVV) (version 0.10)
- Très proche de SVE
- Encore plus souple, la taille des **vecteurs** peut être **redimensionnée** en cours d'exécution !!
- `LMUL`: **coefficient multiplicateur d'une instruction vectorielle**, *unrolling* matériel
- Pas encore d'implémentation matérielle disponible (où très rare)
- Des simulateurs : Spike (fonctionnel), GEM5 (cycle accurate)

Bonus : SIMD dans les consoles de jeux

- Playstation 1 : MIPS + Geometry Transformation Engine pour le calcul vectoriel
- Playstation 2 : Emotion engine 128-bit
- Playstation 3 : IBM CELL (PowerPC, SPU, 128-bit)
- Playstation 4 : AMD Jaguar (AVX1 ou SSE 128-bit)
- Playstation 5 : AMD Zen2 (AVX2 256-bit)

Bonus : et Apple dans tout cela ?

4 Le SIMD dans les systèmes embarqués

- Remplacement des processeurs Intel pour des processeurs de type... ARM!
- Dérivé de l'architecture Firestorm (A14) sur smartphone
- Supporte NEON 2 128-bit !
- Oui mais... Il y en a 4 par cœur → 512 bits en parallèle!

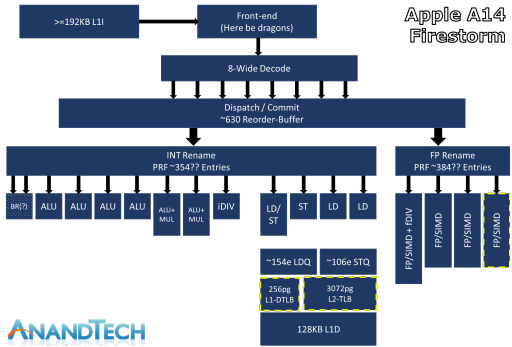


Table des matières

5 Défis, perspectives et conclusion

Les défis applicatifs du SIMD : quelques exemples

- **Big data**
  - On stocke de moins en moins de données brutes mais des données calculées (tout simplement par soucis d'espace de stockage insuffisant)
  - Les calculs sont généralement répétitifs sur différentes données → SIMD
- **Deep learning**
  - Les CPUs SIMD sont souvent battus par les GPUs
  - Regain d'intérêt pour le CPUs pour certains réseaux et des nouvelles instructions dédiées (x86 Advanced Matrix Extension (AMX))
- **Flot optique embarquée**
  - Être capable de traiter un flux vidéo en temps réel pour détecter des objets
  - Tout en limitant la consommation d'énergie (dans un drone par exemple)

Les défis applicatifs du SIMD : bottom-up

Il y a un fort besoin d'experts SIMD à différents niveaux, pour

- Concevoir des **architectures SIMD adaptées**
- **Améliorer les compilateurs** et vectoriser automatiquement du code
- Proposer des bibliothèques légères pour **simplifier la programmation SIMD** (et maximiser la portabilité)
- **Revoir les algorithmes** et les adapter au paradigme SIMD

Conclusion

5 Défis, perspectives et conclusion

- Les instructions SIMD sont **présentes quasiment partout**
- **La performance crête des CPUs** dépend directement des instructions SIMD
- Bien utilisées elles permettent d'atteindre un **haut niveau de performance**
- Mais cela reste bien souvent l'**affaire d'experts**...

Q&R

Merci pour votre écoute !
Avez-vous des questions ?