Cours 7 : la « vectorisation »

Charles Bouillaguet
charles.bouillaguet@lip6.fr

2020-04-16

Principe

La vectorisation, c'est...

- Reformuler les algo...
- pour qu'ils fassent des opérations sur des **vecteurs**

$$A \leftarrow \lambda B + \mu C \qquad A[i] \leftarrow \lambda B[i] + \mu C[i]$$

$$A \leftarrow B + C \times D \qquad A[i] \leftarrow B[i] + C[i] \times D[i] \quad (\textit{fused multiply-add})$$

$$A \leftarrow |B| \qquad A[i] \leftarrow |B[i]|$$

$$A \leftarrow \max(B, C) \qquad A[i] \leftarrow \max(B[i], C[i])$$

$$A[B] \leftarrow C \qquad A[B[i]] \leftarrow C[i] \quad (\textit{scatter})$$

$$A \leftarrow B[C] \qquad A[i] \leftarrow C[B[i]] \quad (\textit{gather})$$

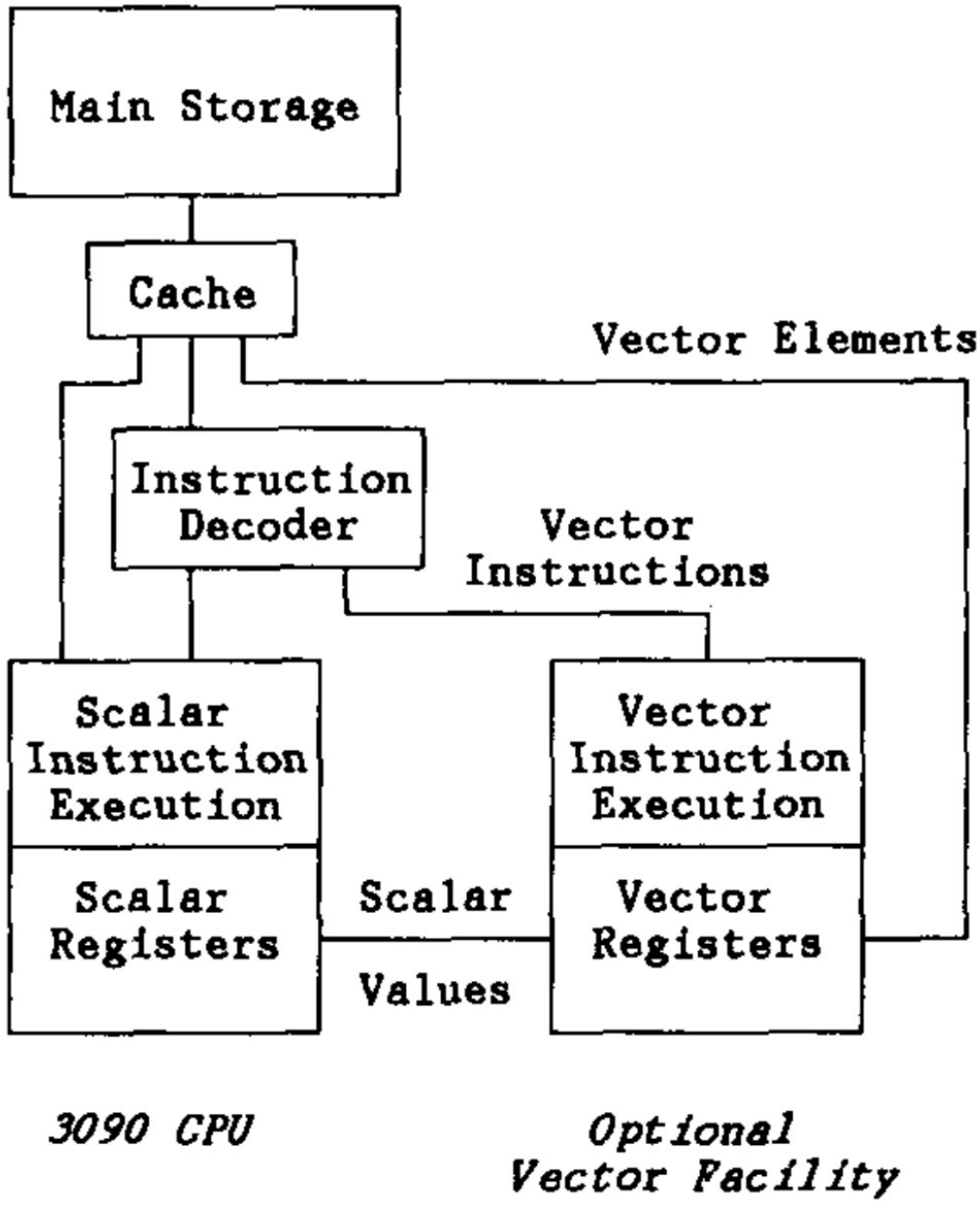
$$A \leftarrow B \leq C \qquad A[i] \leftarrow B[i] \leq C[i]$$

Pourquoi ?

Les ops. sur les vecteurs sont intrinsèquement parallèles

- Permet d'exprimer du parallélisme
  - Fortran, `numpy` gèrent des vecteurs nativement
  - Mais pas le C...
- Il a existé des processeurs vectoriels
  - Cray X1, Cray X-MP, NEC SX, IBM 3090, ...

IBM 3090 CPU


Main Storage
Cache
Vector Elements
Instruction Decoder
Vector Instructions
Scalar Instruction Execution
Vector Instruction Execution
Scalar Registers
Scalar Values
Vector Registers
3090 CPU
Optional Vector Facility

Intérêt

- Possibilité de traitement parallèle
  - bas-de-gamme = un peu, haut-de-gamme = beaucoup
- Une seule instruction donne beaucoup de travail
  - Économies sur le décodage des instructions
- Opérations enchainées $\rightsquigarrow$ pipelining
  - Efficacité énergétique

Mais pourquoi ça a disparu ?

- Nécessite mémoire rapide pour transférer les vecteurs
- Scatter/gather sont particulièrement horribles
- $\Rightarrow$ processeurs/machines *dédiés* au HPC
- Mais tendance « commodity components » (ASCI red)

Le présent : instructions SIMD

1985 IBM 3090, registres vectoriels de $32Z$ bits

- Z pas spécifié, *implementation-dependant*
- Dans les faits, $Z = 128$, donc registres de 4096 bits

Le présent : instructions SIMD

1985 IBM 3090, registres vectoriels de $32Z$ bits

- Z pas spécifié, *implementation-dependant*
- Dans les faits, $Z = 128$, donc registres de 4096 bits

Le présent : instructions SIMD

1985 IBM 3090, registres vectoriels de $32Z$ bits

- Z pas spécifié, *implementation-dependant*
- Dans les faits, $Z = 128$, donc registres de 4096 bits

1996 Intel Pentium MMX : vecteurs de 64 bits (`8*int8/4*int16`)

Le présent : instructions SIMD

1985 IBM 3090, registres vectoriels de $32Z$ bits

- Z pas spécifié, *implementation-dependant*
- Dans les faits, $Z = 128$, donc registres de 4096 bits

1996 Intel Pentium MMX : vecteurs de 64 bits (`8*int8/4*int16`)

1999 Intel SSE : vecteurs de 128 bits, `4*float`

2001 Intel SSE2 : vecteurs de 128 bits, `int` et `double`

2004 Intel SSE3 : plus d'opérations

2007 Intel SSE4 : plus d'opérations

2010 Intel AVX : vecteurs de 256 bits, `8*float`

2010 Intel AVX2 : vecteurs de 256 bits, `int` et `double` + `gather`

2017 Intel AVX512 : 512 bits, tous types + `scatter/gather/mask`

ISA	Accronyme	Nom	année	# regs	|regs|
IBM 3090		Vector Facility	1985	16	variable
x86-64	MMX	Multi Media eXtensions	1996	8	64
	3DNow !		1998	8	64
	SSE SSE2 SSE3 SSSE3 SSE4	Streaming SIMD Extensions	1999 2001 2004 2004 2007	16	128
	AVX AVX2	Advanced Vector eXtensions	2010 2013	16	256
	AVX512		2017	32	512
PowerPC	Altivec		1999	32	128
ARM v7	NEON	Advanced SIMD extension	2005 2011	16 32	128
ARM v8					
	SVE	Scalable Vector Extensions	2020	32	variable

- ARMv8 SVE : registres de 128–2048 bits
- `fugaku` implante SVE avec des registres des 512 bits
- `RISC-V` a une extension vectorielle inspirée d'IBM 3090

Deux styles

« Vraies » architectures vectorielles à l'ancienne

- Taille des registres inconnue, voire pas de registres du tout
- Fonctionnalités : Scatter + Gather + masques
- **Revival** : ARMv8 SVE, RISC-V, AVX-512, ...

Instructions SIMD disponibles dans les CPU grand public

- Taille des registres fixée
- Empilement de générations successives
- Accès à la mémoire contraint

Entre les deux/à côté : les GPUs

- ARM Neon est dans tous les smartphones
- Presque tous les CPUs x86 ont AVX2 aujourd'hui
- AVX512 : pas pour tout le monde
  - Xeon bronze/silver/gold depuis 2017
  - Certains core i3/i5/i7 très récents (mais pas tous...)
- Ici : `ppti-gpu-[1,4,5]` ont l'AVX512
- Grid'5000 : certains clusters l'ont.

```
bouillaguet@ppti-gpu-1:~$ cat /proc/cpuinfo
...
model name      : Intel(R) Xeon(R) Gold 6152 CPU @ 2.10GHz
...
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts
rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq
dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid
dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx
f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cdp_l3
invpcid_single pti intel_ppin ssbd mba ibrs ibpb stibp tpr_shadow vnmi
flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2
erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap
clflushopt clwb intel_pt avx512cd avx512bw avx512vl xsaveopt xsavec
xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm
ida arat pln pts pku ospke flush_l1d
```

Problèmes avec les jeux d'instructions SIMD

Leur multiplicité les rend difficiles à utiliser

- Nécessite recompilation à chaque nouvelle version
  - Voire réécriture du code / changement des algos...
- Rend le code non-portable
  - Le CPU qui va exécuter le programme a-t-il AVX-xxx ?
  - Déterminer à l'exécution le CPU...
  - ... Et utiliser la fonction appropriée (soupir)

Comment utiliser les instructions SIMD/vectorielles ?

1. Écrire de l'assembleur
2. Utiliser les « *instrinsics* » des compilateurs
   - Pseudo-fonctions qui émettent une instruction SIMD
3. Laisser faire le compilateur (vectorisation automatique)
4. Donner des directives de vectorisation (OpenMP $\geq$ v4.0)

La vectorisation en pratique

Principe général

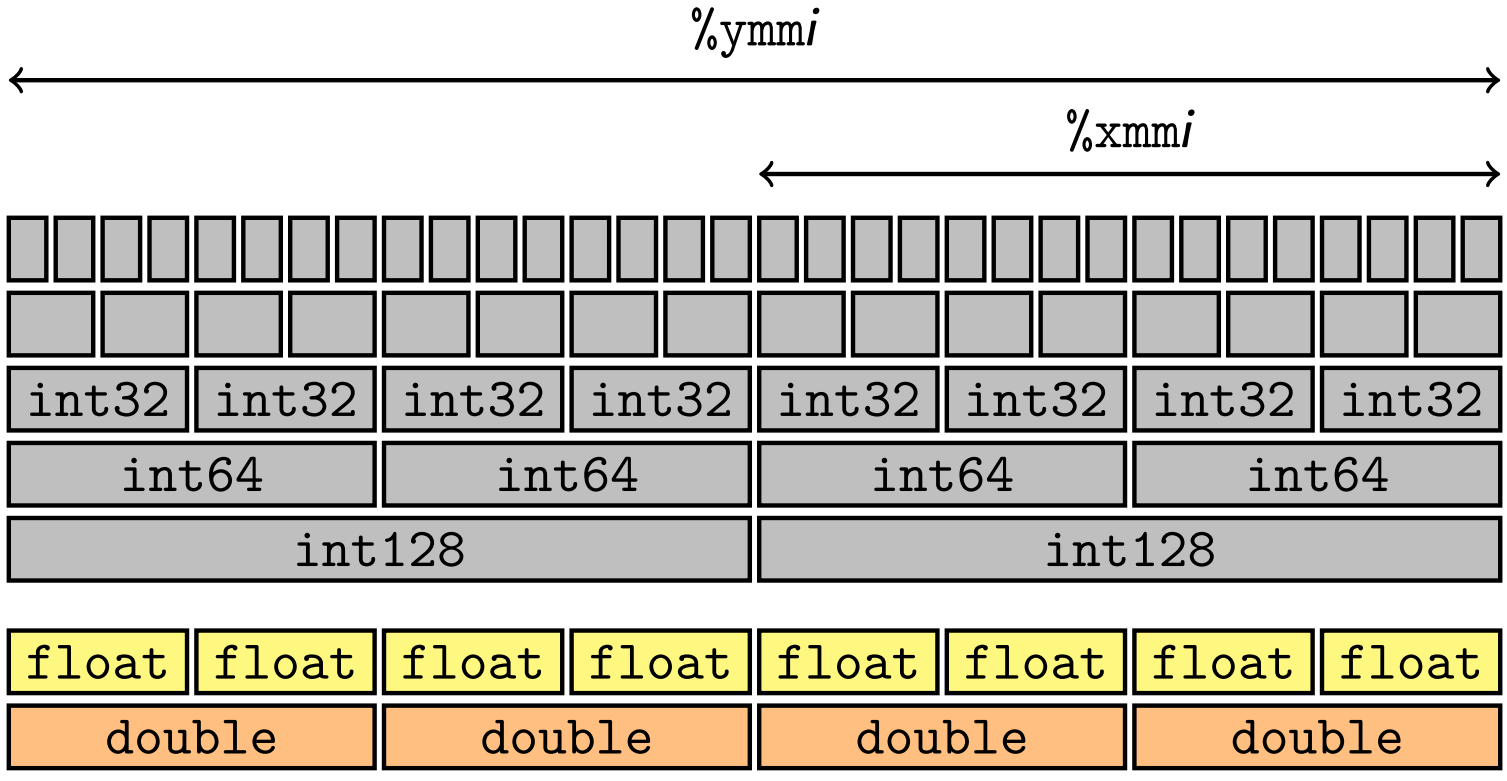
1. Charger les données dans registres SIMD
2. Calculer avec les instructions SIMD
3. Écrire le résultat en mémoire

Conditions d'utilisation

- Se déroule à **l'intérieur d'un thread**.
  - Parallélisation = répartir sur plusieurs coeurs
  - Vectorisation = utiliser instructions SIMD dans UN coeur
- La vectorisation concerne des **portions de code réduites**
  - **Boucles** simples (limites du jeu d'instruction)

  $\Rightarrow$ cible généralement les boucles les **plus internes**

SSEx + AVX + AVX2



Instructions	registres	taille
SSE*x*	%xmm0, ..., %xmm15	128 bits
AVX + AVX2	%ymm0, ..., %ymm15	256 bits
AVX512	%zmm0, ..., %zmm31 %k0, ..., %k7	512 bits 64 bits

Nomenclature des instructions

Les instructions SIMD flottantes ont des noms composés :

`<prefix><op><simd or not><type>`

où :

- `<prefix>` : vide (SSE), `v` (AVX, AVX2, AVX512)
- `<op>` `add`, `sub`, `mul`, `div`, `fmadd`, `min`, `max`, `abs`, `floor`, `ceil`, `round`, ...
- `<simd or not>` `s` (*scalar*), `p` (*packed* — c.a.d. vecteur).
- `<type>` `s` (*single*, `float`), `d` (`double`)

E.g. `vmaxpd`, `vmulss`, etc.

Instructions SIMD dans le compilateur gcc

- La **vectorisation automatique** est désactivée par défaut
  - Par défaut, GCC compile au niveau d'optimisation zéro
  - Vectorisation automatique et via OpenMP sont désactivées
- On peut l'activer par l'option `-ftree-vectorize`
  - `-O3` entraine cette option
- On fait du HPC : compiler avec `-O2` ou `-O3` !

```
~$ gcc -O1 -ftree-vectorize vectorisation_auto.c
~$ gcc -O1 -fopenmp vectorisation_omp.c
```

- gcc n'émet **pas** les instructions AVX2 (resp. FMA) par défaut
  - Ajouter options `-mavx2` (resp. `-mfma`).
- Verbosité et diagnostic :
  - `-fopt-info-vec`, `-fopt-info-vec-missed` et `-fopt-info-vec-all`

Vectorisation automatique

- Fortran a des vecteurs explicites
  - Facile à vectoriser
- En C, le compilateur détecte les **boucles vectorisables**

Code C

```
for (int i = 0; i < n; i++)
        C[i] = lambda * A[i] + mu * B[i];
```

Code assembleur produit par gcc

```
.loop:
        vmulpd   (%rsi,%rax), %ymm3, %ymm0
        vmulpd   (%rcx,%rax), %ymm2, %ymm1
        vaddpd   %ymm1, %ymm0, %ymm0
        vmovapd  %ymm0, (%rdx,%rax)
        addq     $32, %rax
        cmpq     %rcx, %rax
        jne      .loop
```

Obstacles à la vectorisation (automatique ou pas)

- Dépendance de donnée (itérations pas indépendantes)

```
for (int i = 1; i < n; i++)
    a[i] += a[i-1];
```

- `if` (traitement différencié selon les cases du vecteur)
- Accès à la mémoire compliqués et/ou pb. d'alignement
- Calculs trop compliqués (sin, cos, log, exp, ...)

Problèmes d'alignement

Règle générale

Vecteur de *xxx* octets doit être à une adresse multiple de *xxx*

Solutions :

- `double A[n] __attribute__ ((aligned(32)));`
  - Ceci est du code C correct !
- `int posix_memalign(void **ptr, size_t align, size_t size);`
- `void *aligned_alloc(size_t alignment, size_t size);`
  - Inclus dans le langage C11

Problèmes d'accès compliqués à la mémoire

Array of struct

```
struct point {
      double x, y, z;
};

struct point *A;
```

xyzxyzxyzxyzxyzxyz....

Struct of Array

```
struct points {
      double *x, *y, *z;
};
struct points A;          // potentiellement plus facile à vectoriser
```

xxxxxxxxxxxxxx....

yyyyyyyyyyyyyy....

zzzzzzzzzzzzz....

Vectorisation de boucles simples : *strip-mining*

Code original

```
for (int i = 0; i < n; i++)
    u[i] = u[i] + alpha * v[i];
```

Code transformé

```
int m = n - (n % 4);
/* traitement par lots de 4 avec instructions SIMD */
for (i = 0; i != m; i += 4) {
    u[i + 0] = u[i + 0] + alpha * v[i + 0];
    u[i + 1] = u[i + 1] + alpha * v[i + 1];
    u[i + 2] = u[i + 2] + alpha * v[i + 2];
    u[i + 3] = u[i + 3] + alpha * v[i + 3];
}
/* épilogue au cas où n ne soit pas un multiple de 4 */
for (int i = m; i < n; i ++)
    u[i] = u[i] + alpha * v[i];
```

Directive OpenMP simd

Rappel : nécessite l'option `-fopenmp`

```
#pragma omp simd
for (int i = 0; i < n; i++)
   ...
```

- Regroupe les itérations en *chunks* de taille *N*
  - effectue le calcul avec une unité SIMD à *N* éléments
- On « promet » au compilateur que la boucle est vectorisable
- Pas de parallélisation multi-thread

Directive OpenMP simd

```
#pragma omp simd [clause[[,] clause],...]
for (int i = 0; i < n; i++)
    ...
```

Clauses possibles (list non-exhaustive)

- `reduction(+:v)` : déjà connu
  - Casse dépendance de données sur la variable qui accumule
- `simdlen(length)` : taille de *chunk* souhaitée
- `safelen(length)` : taille de *chunk* maximale
  - E.g., au-delà, un problème de dépendance pourrait arriver
- `aligned(v[:k])` : promet que *v* est alignée sur `k` octets
- `linear(x[:step])` : *x* dépend de *i* (`x_i = x_init + i * step;`)

Utilisations des *intrinsics*

Déclarations :

```
#include <xmmintrin.h> SSE
#include <emmintrin.h> SSE2
#include <pmmintrin.h> SSE3
#include <tmmintrin.h> SSSE3
#include <smmintrin.h> SSE4.1
#include <nmmintrin.h> SSE4.2
#include <immintrin.h> AVX & AVX2
#include <zmmintrin.h> AVX512
```

Types de données pour les *intrinsics*

`__m256`	256 bits (8 `float`)
`__m256d`	256 bits (4 `double`)
`__m256i`	256 bits (entiers)
`__m128`	128 bits (4 `float`)
`__m128d`	128 bits (2 `double`)
`__m128i`	128 bits (entiers)