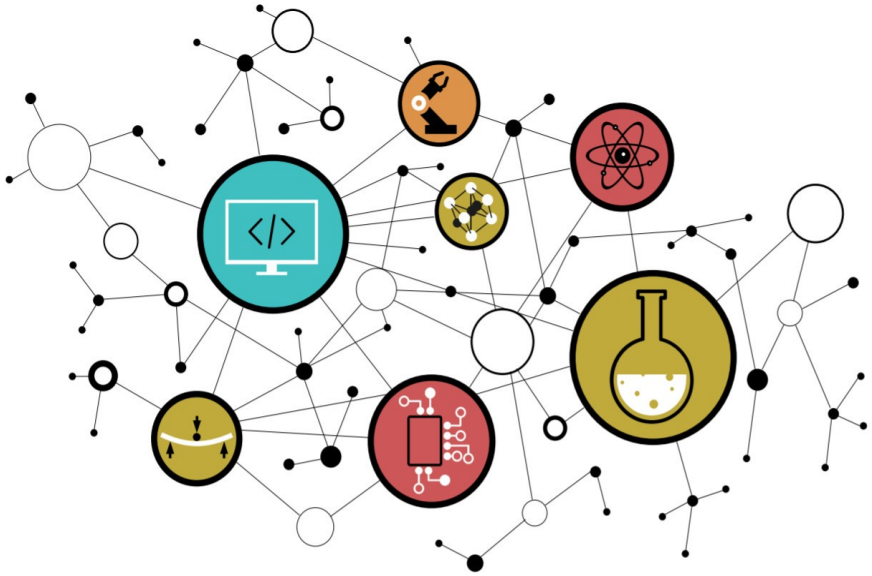
TRAVAUX PRATIQUES

PROGRAMMATION VECTORIELLE SUR GPP INTEL x86_64



Hugo Descoubes – neku – GNU\Linux - Ubuntu 20.04 LTS – LibreOffice 6.4.6.2 – 2021

SOMMAIRE

3.1. Analyse du programme de test

Nous allons effectuer un Benchmarking entre architectures processeurs en mettant en confrontation une architecture superscalaire GPP (General Purpose Processor) Intel corei7 en micro-architecture Haswell cadencée à 3,6GHz (machines de TP en salles A203/A201) et notre architecture DSP VLIW C6600 cadencée à 1,4GHz proposée par Texas Instruments. Notre première comparaison se fera sur une implémentation en C canonique.

Intel corei7 – 4790 Haswell 4th gen	**Texas Instruments TMS320C6678**
3,6GHz, 105W en charge prix unitaire : 325€ (en 2015)	1,4GHz, 10W en charge prix unitaire : 240€ (160€ pour 1Ku)

- Ouvrir un shell et se déplacer dans le répertoire de travail disco/ia64. Afficher et analyser le contenu du fichier README.md à la racine. Compiler le projet et analyser les fichiers de test et résultats.

```
<your_computer_path>/disco/ia64$ make
<your_computer_path>/disco/ia64$ ./build/bin/firtest
```

- Exécuter plusieurs fois d'affilée le programme de test et observer les mesures de l'algorithme. Que constatez-vous ?

Nous pouvons constater qu'une implémentation en C canonique, même avec déroulement de boucle, garantit une portabilité du code, même sur des architectures matérielles différentes. Nous constatons également, que contrairement au CPU VLIW d'un DSP C6600, les CPU superscalaires des GPP x86_64 ne sont pas déterministes et donc ''potentiellement'' moins adaptés aux applications temps réel imposant des contraintes dures. Ceci est principalement lié aux étages suivants :

- Technologie du pipeline superscalaire (exécution Out-Of-Order, prédiction, spéculation, etc)
- Utilisation d'un modèle mémoire pleinement cachable pour les niveaux L1/L2/L3
- Utilisation d'une MMU (Memory Management Unit) pour la translation d'adresses virtuelles en adresses physiques (Table de translation, TLB, etc)
- Au côté multi-applicatif du système de la machine de test. De plus, Linux comme le noyau de Windows ne sont pas des kernels temps réel (scheduler non déterministe). En effet, bien d'autres programmes de même privilège sont en cours d'exécution sur l'ordinateur de développement (exécuter l'utilitaire htop pour observer en temps réel les processus et threads en cours d'exécution)

- Dans le Makefile, nous pouvons observer que des options de compilation spécifiques sont passées à GCC. A quoi correspondent les 3 options suivantes (s'aider de *man gcc* et d'internet):
    - -03
    - -std=c99
    - -march=native

```
CFLAGS = -Wall -march=native -std=c99 -O3
```

- Dans le fichier disco/ia64/test/src/main.c, nous pouvons observer la définition d'une fonction inline, dans notre cas implémentée en C avec insertion d'une séquence en assembleur 64bits x86_64. Cette fonction force le CPU courant à vider son pipeline matériel puis réalise une lecture de la valeur courante du core timer TSC (timer présent dans chaque CPU), comme précédemment sur architecture C6600. Sur architectures compatibles x86-x64, ce timer 64bits est démarré à la mise sous tension de la machine et compte jusqu'à débordement. En vous aidant d'internet, rappeler le rôle du spécificateur inline, préciser l'intérêt et donner des exemples d'utilisation ?

```
inline unsigned long long __attribute__((always_inline)) rdtsc_inline()
{
      unsigned int hi, lo;
      __asm__ __volatile__(
            // flush core pipeline
            "xorl %%eax, %%eax\n\t"
            "cpuid\n\t"
            // read current TSC value
            "rdtsc"
                  : "=a"(lo), "=d"(hi)
                  : // no parameters
                  : "rbx", "rcx");
      return ((unsigned long long)hi << 32ull) | (unsigned long long)lo;
}

...


start = rdtsc_inline();
fir_sp (xk_sp, a_sp, yk_sp_cn, A_LENGTH, YK_LENGTH);
stop = rdtsc_inline();
duration = stop - start;

...
```

- Quelles sont les principales différences entre une fonction inline et une fonction intrinsèque (s'aider d'internet)?

- Reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude. Prendre une moyenne de 10 mesures. Vous pouvez programmer la boucle de test si vous le souhaitez.

3.2. Vectorisation avec ISA extension SSE4.1

Nous allons maintenant comparer ce qui peut être comparable, à savoir les performances de notre code vectorisé sur architecture C6600 à du code vectorisé sur architecture IA-64. Nous nous intéresserons notamment à l'extension vectorielle SIMD SSE4.1 proposée par Intel. Pour information, courant 2014, Intel proposa sur sa micro-architecture Haswell une extension DSP (Digital Signal Processing). Cette extension, nommée FMA (Fused Multiply-Add), est donc dédiée aux applications de traitement numérique du signal mais ne sera néanmoins pas abordée en travaux pratiques. Pour les plus curieux, ne pas hésiter à aller voir sur MSDN (MicroSoft Developer Network) les quelques fonctions intrinsèques proposées.

Cette partie n'a pas vocation à permettre de découvrir en profondeur l'architecture interne des processeurs compatibles x86_64 (Intel ou AMD), notamment les architectures Intel. Néanmoins, nous allons pouvoir constater que nos précédents acquis nous permettent maintenant d'effectuer de la vectorisation de code sur toute architecture vectorielle processeur. Les concepts resteront le plus souvent les mêmes. Avant tout, nous avons à savoir que les architectures x86_64 actuelles possèdent plusieurs banques de registres vectoriels :

- Registres MMX 64bits : peuvent contenir jusqu'à 2 flottants 32bits
- Registres XMM 128bits : peuvent contenir jusqu'à 4 flottants 32bits
- Registres YMM 256bits : peuvent contenir jusqu'à 8 flottants 32bits

Dans notre cas, les instructions de l'extension SSE4.1 travaillent avec les registres XMM 128bits et sont aptes à manipuler les flottants en simple précision (IEEE-754) par paquets de 4. Le type conteneur __m128 permet donc de stocker jusqu'à 4 flottants en simple précision :

```
Container type (4 x 32bits floating point) : __m128
```

https://msdn.microsoft.com/fr-fr/library/y0dh78ez(v=vs.90).aspx

- Observons les préfixations et suffixations des fonctions intrinsèques présentées dans la suite de cet exercice :

```
Intel x86_64 Intrinsic syntax: _mm_<intrinsic_name>_<data_type>
```

- Mise à zéro des éléments d'un vecteur de flottants. Pour information, ps signifie Packet Single, soit paquet de flottants en simple précision 32bits IEEE754 :

```
__m128  float_vector;

 float_vector = _mm_setzero_ps ();
```

https://msdn.microsoft.com/fr-fr/library/tk1t2tbz(v=VS.90).aspx

- Initialise les éléments d'un vecteur de flottants :

```
__m128  float_vector;

 float_vector = _mm_set_ps (1.0, 2.2, 3.0, 7.0);
```

https://msdn.microsoft.com/fr-fr/library/afh0zf75(v=vs.90).aspx

- Chargement d'un vecteur de 4 flottants alignés depuis la mémoire vers un registre XMM de destination :

```
float  tab[4] = {1.0, 3.4, 5.0, 6.0};

__m128  float_vector;

 float_vector = _mm_load_ps (&tab[0]);
```

https://msdn.microsoft.com/fr-fr/library/zzd50xxt(v=vs.90).aspx

- Chargement d'un vecteur de 4 flottants non-alignés depuis la mémoire vers un registre XMM de destination :

```
float  tab[4] = {1.0, 3.4, 5.0, 6.0};

__m128  float_vector;

 float_vector = _mm_loadu_ps (&tab[0]);
```

https://msdn.microsoft.com/fr-fr/library/x1b16s7z(v=vs.90).aspx

- Produit scalaire entre deux vecteurs de 4 flottants (result = a0.x0 + a1.x1 + a2.x2 + a3.x3). Le résultat du produit scalaire est un nombre scalaire, le résultat est par défaut sauvé dans les 32bits de poids faible du vecteur 128bits de destination:

```
__m128  float_vector_src1;
__m128  float_vector_src2;
__m128  float_vector_dst;

float_vector_dst = _mm_dp_ps (float_vector_src1, float_vector_src2, 0xff);
```

https://msdn.microsoft.com/fr-fr/library/bb514054(v=vs.90).aspx

- Addition de deux vecteurs contenant 4 flottants chacun :

```
__m128  float_vector_src1;
__m128  float_vector_src2;
__m128  float_vector_dst;

 float_vector_dst = _mm_add_ps (float_vector_src1, float_vector_src2);
```

https://msdn.microsoft.com/fr-fr/library/c9848chc(v=vs.90).aspx

- Sauvegarde en mémoire d'un vecteur de 4 flottants alignés :

```
float  tab[4];
__m128  float_vector;

float_vector = _mm_set_ps (1.0, 2.2, 3.0, 7.0);

_mm_store_ps (&tab[0], float_vector);
```

https://msdn.microsoft.com/fr-fr/library/s3h4ay6y(v=vs.90).aspx

- Ouvrir le fichier /disco/ia64/firlib/h/firlib.h et observer les déclarations de type et d'union présentées ci-dessous. Cette union, greffée à une déclaration de type, permet, après déclaration d'une variable conteneur, d'accéder à un vecteur XMM soit élément par élément, soit directement au vecteur 128bits complet. En vous aidant d'internet, rappeler ce qu'est une union et préciser la différence avec une structure ?

```
union xmm_t {
      __m128   m128_vec;      // full access to XMM vector
      float    m128_f32[4];   // access to XMM vector elements
} align16_xmm;

typedef union xmm_t xmm_t;
```

- Dans l'exemple ci-dessous, comment accéder au 3ième élément de la variable vectorielle data_vec de type conteneur XMM 128bits.

```
xmm_t   data_vec;
```

- Ouvrir le fichier /disco/ia64/firlib/src/fir_sp_sse_r4.c puis implémenter le code vectorisé pour architecture x64 correspondant à la fonction fir_sp_r4. Valider son bon fonctionnement sans oublier de vérifier l'alignement mémoire des différents vecteurs de données traités par l'algorithme.

- Reporter les résultats des tests dans le tableau d'analyse comparative ou Benchmarking présent dans le document de Prélude. Prendre une moyenne de 10 mesures. Vous pouvez programmer la boucle de test si vous le souhaitez.

3.3. Synthèse

Rappelons les familles d'architectures CPU rencontrées sur le marché, et le plus souvent les familles de processeurs associées :

- CPU à pipeline classique (code In-Order en mémoire et exécution In-Order):
  - Rencontré par exemple sur MCU et certains DSP (processeur faible coût, consommation et encombrement réduits). Dans l'exemple des MCU (RISC-V, ARM Cortex-M, STM32 STMicroelectronics, PIC Microchip, Intel 8051, etc), le processeur embarque une application pouvant être sans OS (Baremetal) ou avec un système d'exploitation temps réel léger ou RTOS (FreeRTOS, Zephyr, etc).
  - Ces CPU peuvent être sinon intégrés en grand nombre sur certains processeurs, prenons l'exemple des GPU. Dans le cas des GPU, applications de calcul massivement parallèle (traitement d'image, vidéo, graphique 3D, finances, etc)
- CPU à pipeline superscalaire (code In-Order en mémoire et exécution Out-Of-Order) : grande polyvalence (processeur généraliste Système/Application/Calcul), le plus souvent grande puissance de calcul, pipeline complexe (consommation), donc forte intelligence déportée dans chaque cœur avec un faible niveau de parallélisme (typiquement 2, 4, 8, 16 cœurs souvent vectoriels). Rapport performance/consommation faiblement intéressant (par rapport aux MPPA, DSP, FPGA, etc). Chaque cœur (ensemble CPU et caches locaux) implémente le plus souvent les mécanismes suivants :
  - Prédiction au branchement
  - Étage d'exécution Out-Of-Order (exécution spéculative, étage de retirement avec étage de renommage des registres, etc)
  - Plusieurs unités d'exécution vectorielles par CPU aptes à travailler en parallèle
  - Mécanismes d'accélération aux étages de décodage, capture de boucles, etc
  - Cependant, dû à la complexité du pipeline, ces architectures peuvent offrir quelques irrégularités au regard du déterminisme à l'exécution (x86, x64, ARM cortex-A, IBM/APPLE/Freescale PowerPC, MIPS Aptiv, etc).
- CPU à pipeline VLIW ou EPIC (code Out-Of-Order en mémoire et exécution In-Order): forte puissance de calcul, pipeline relativement simple en opposition aux architectures superscalaires, donc rapport performance/consommation intéressant. Intelligence et compétences déportées vers le développeur et la toolchain. Grand déterminisme à l'exécution. Néanmoins, développements dépendants de l'architecture nécessaires, problèmes de portabilité de code. Rétrocompatibilité au niveau binaire difficile à suivre pour les fondeurs (MPPA Kalray, DSP TI C6000, NXP TriMedia, DSP SHARC Analog Device, ST200 STMicroelectronics, Intel Itanium, etc).

- Langage C vs Assembleur : dans une optique d'optimisation, un développement en langage C avec une bonne connaissance de l'architecture matérielle, du jeu d'instructions et des mécanismes d'optimisation de la toolchain peut éviter un passage à l'étage assembleur à notre époque (intrinsics, directives de compilation, déroulement de boucle, etc) et donc accélérer le TTM du produit (Time To Market). De même, nous avons effectué nos compilations sous gcc. Si nous souhaitons gagner en performance, il serait alors intéressant de travailler directement avec les outils fondeurs. Par exemple, si nous souhaitons garantir des performances optimales sur architecture Intel, il nous faudrait alors utiliser icc (Intel C++ Compiler).

- Tests : ne pas négliger les procédures de test (conformité et performance), notamment dans une optique de Benchmarking et d'optimisation d'algorithmes.


notes