TP Architechture et optimisation de code

M2 CHPS

Optimisation de code Mask

Auteurs :
Mme Lamia Dendani

Enseignants référents :
M. Emmanuel oseret
M. Salah Ibnamar
M. William Jalby

Université Paris-Saclay

30 novembre 2023

1 Introduction

Le TP noté a pour sujet l'optimisation d'un code en C avec l'aide de l'outil **MAQAO**. Il s'agit de faire une analyse des performances du programme **mask.c** en séquentiel. Premièrement, après avoir exposé les caractéristiques de l'architecture ciblée et de ce qu'elle supporte, nous présentons les performances du code de base en établissant une référence sur laquelle nous appuyer pour la suite. Deuxièmement, l'analyse du code **mask** en parallel à l'aide des primitive d'**OpenMP**.Enfin, nous présentons les différentes optimisations effectuées sur le code ainsi que les performances obtenues en séquentiel ainsi qu'en parallèle en fonction du compilateur choisis ainsi que le nombre de threads pour la version parallèle. lien Github : https ://github.com/lamialami1997/mask

2 Architecture matériel

Le CPU sur lequel est exécuté le code à optimiser est un Intel(R) Xeon(R) CPU E5-2670 0 @2.60GHz, composé de 8 coeurs physiques, chacun constitué de 2 coeurs logiques (32 coeurs de calcul au total). Il supporte au mieux de l'AVX2 et SSE , et d'une fréquence de 1754.6 MHz.Les tailles de caches L1, L2, L3 sont respectivement 512KiB, 4MiB et 40MiB(cf Figure 1).

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          32
On-line CPU(s) list:             0-31
Thread(s) per core:              2
Core(s) per socket:              8
Socket(s):                       2
NUMA node(s):                    2
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           45
Model name:                      Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz
Stepping:                        7
CPU MHz:                         1754.599
CPU max MHz:                     3300.0000
CPU min MHz:                     1200.0000
BogoMIPS:                        5187.91
Virtualization:                  VT-x
L1d cache:                       512 KiB
L1i cache:                       512 KiB
L2 cache:                        4 MiB
L3 cache:                        40 MiB
NUMA node0 CPU(s):               0-7,16-23
NUMA node1 CPU(s):               8-15,24-31
Vulnerability Itlb multihit:     KVM: Vulnerable
Vulnerability L1tf:              Mitigation; PTE Inversion
Vulnerability Mds:               Mitigation; Clear CPU buffers; SMT vulnerable
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Mmio stale data:   Unknown: No mitigations
Vulnerability Retbleed:          Not affected
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Retpolines, IBPB conditional, IBRS_FW, STIBP conditional, RS
                                 B filling, PBRSB-eIBRS Not affected
Vulnerability Srbds:             Not affected
Vulnerability Tsx async abort:   Not affected
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 rdtsc
                                 p lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_
                                 tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2
                                  ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_
                                 timer aes xsave avx lahf_lm epb pti ssbd ibrs ibpb stibp tpr_shadow vnmi
                                  flexpriority ept vpid xsaveopt dtherm ida arat pln pts md_clear flush_l
                                 1d
```

FIGURE 1 – Caractéristiques de la machine

3 Code de base et analyse avec MAQAO

Le code fourni est constitué d'un fichier `mask.c` qui permet d'effectuer une opération XOR bit à bit entre les valeurs de "a" et "b" et de stocker le résultat dans "c".

Algorithme 1 : Fonction `mask()`

```
void mask(const u8 *restrict a, const u8 *restrict b, u8 *restrict c, u64 n)
{
  for (u64 i = 0; i < n; i++)
    c[i] = a[i] ^ b[i];
}
```

3.1 Compilation et exécution du code

1. Compilation et exécution du code `mask` : `make` pour la compilation et pour l'exécution il suffit de lancer le script shell `exec.sh` suivant :

`exec.sh`

```
#!/bin/bash
./genseq s1.dna 10000000
./genseq s2.dna 10000000
echo -e "GCC :\n"
./mask.g s1.dna s2.dna
echo -e "ICX :\n"
./mask.i s1.dna s2.dna
```

2. Lancer avec l'outil MAQAO :

   maqao oneview –R1 — ./mask.g s1.dna s2.dna

3.2 Première analyse

Les exécutions avec la commande `time` donne un temps aux alentours de 7.4 secondes et MAQAO donne plus précisément un temps de 29.50 secondes (cf Figure 3). C'est notre temps de référence.
Le rapport global sur la Figure 3 nous indique principalement qu'il manque des flags de compilation et que l'efficacité des accès aux tableaux est seulement de 50%. Cela signifie que l'accès aux données ne se fait pas de manière contigu.

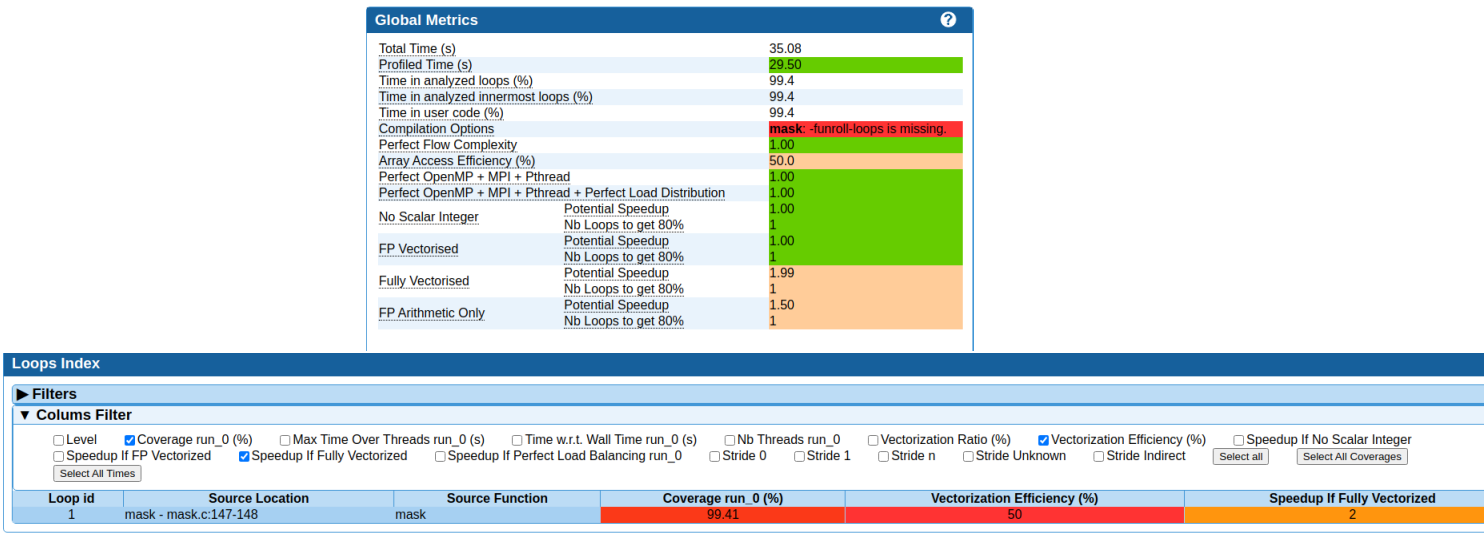

FIGURE 2 – Rapport global MAQAO

Le rapport sur les boucles (cf Figure 3) nous indique que 99.41% de l'exécution du code se fait dans la fonction `mask()`. Il devient plus qu'évident que le travail d'optimisation doit se faire sur cette fonction.

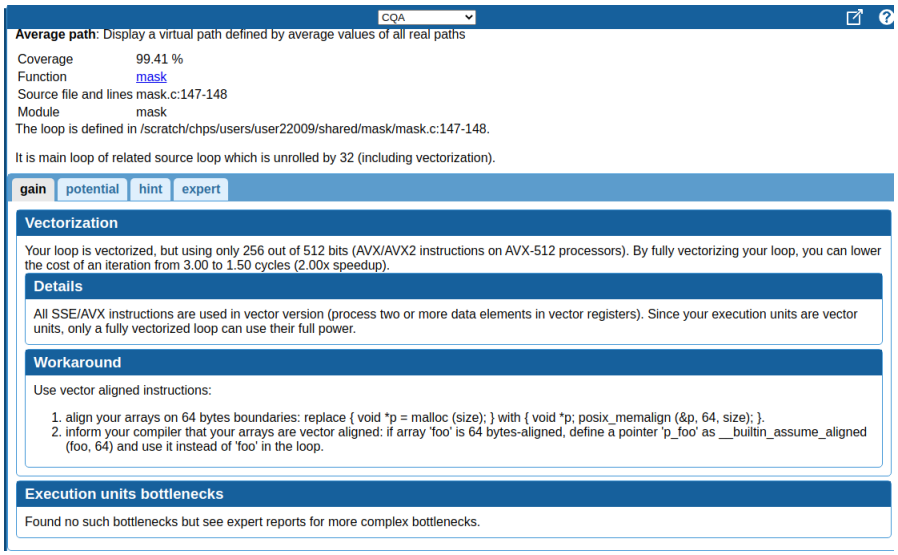

FIGURE 3 – Rapport global MAQAO

4 Optimisations

4.1 Au niveau de la compilation

La première optimisation est l'ajout des flags de compilation pour que la génération de code de `mask.c` s'adapte le plus possible à l'architecture cible. Lors du TP,la plupart des flags d'optimisation nécessaire sont fournis avec le code de bas nous avons donc ajouté les flags suivants :

1. `-march=native` : permet de compiler le code pour la machine spécifique sur laquelle il sera exécuté, en utilisant les instructions spécifiques à cette machine et aide le compilateur à appliquer les optimisations appropriées pour cette architecture, comprend le flag `-mavx2` pour généré des instructions AVX2.
2. `-Ofast` : permet d'activer la vectorisation par défaut via le flag inclus `-O3`, qui lui-même inclut `-finline-functions` permettant d'insérer le code des fonctions appelées directement dans le code appelant, `-ftree-vectorize` et `-ftree-loop-vectorize` permettant la vectorisation automatique des boucles. Lorsque le compilateur trouve une boucle qui est possible à vectoriser, il générera automatiquement du code compatible SIMD. Ce niveau d'optimisation comprend aussi les optimisations des opérations mathématiques, flag `-ffast-math`, ce qui peut induire des erreurs arithmétiques, mais dans notre cas nous avons vérifié que nous n'avons pas affecté les résultats de base.
3. `-funroll-loops` : permet le déroulage de boucle ce qui peut améliorer les performances en réduisant le nombre d'instructions de test et d'incrémentation de l'indice de la boucle déroulée.

4.1.1 Unrolling

Le déroulage de boucle permet d'améliorer les performances du code en réduisant le nombre de sauts dans le code, elle réduit le nombre d'itérations nécessaire pour exécuter la boucle, elle peut également augmenter le parallélisme du code.

Algorithme 3 : Fonction `mask_unroll()`

```
void mask_unroll(const u8 *restrict a, const u8 *restrict b, u8 *restrict c, u64 n)
{
  //
  for (u64 i = 0; i < n - (n%32); i+=32)
    {
      c[i+0] = a[i+0]  ^ b[i+0] ;
      c[i+1] = a[i+1]  ^ b[i+1] ;
      c[i+2] = a[i+2]  ^ b[i+2] ;
      c[i+3] = a[i+3]  ^ b[i+3] ;
      c[i+4] = a[i+4]  ^ b[i+4] ;
      c[i+5] = a[i+5]  ^ b[i+5] ;
      c[i+6] = a[i+6]  ^ b[i+6] ;
      c[i+7] = a[i+7]  ^ b[i+7] ;
      c[i+8] = a[i+8]  ^ b[i+8] ;
      c[i+9] = a[i+0] ^ b[i+0];
      c[i+10] = a[i+10] ^ b[i+10];
      c[i+11] = a[i+11] ^ b[i+11];
      c[i+12] = a[i+12] ^ b[i+12];
      c[i+13] = a[i+13] ^ b[i+13];
      c[i+14] = a[i+14] ^ b[i+14];
      c[i+15] = a[i+15] ^ b[i+15];
      c[i+16] = a[i+16] ^ b[i+16];
      c[i+17] = a[i+17] ^ b[i+17];
      c[i+18] = a[i+18] ^ b[i+18];
      c[i+19] = a[i+19] ^ b[i+19];
      c[i+20] = a[i+20] ^ b[i+20];
      c[i+21] = a[i+21] ^ b[i+21];
      c[i+22] = a[i+22] ^ b[i+22];
      c[i+23] = a[i+23] ^ b[i+23];
      c[i+24] = a[i+24] ^ b[i+24];
      c[i+25] = a[i+25] ^ b[i+25];
      c[i+26] = a[i+26] ^ b[i+26];
      c[i+27] = a[i+27] ^ b[i+27];
      c[i+28] = a[i+28] ^ b[i+28];
      c[i+29] = a[i+29] ^ b[i+29];
      c[i+30] = a[i+30] ^ b[i+30];
      c[i+31] = a[i+31] ^ b[i+31];
    }
    for (u64 i = n - (n%32); i < n; i++)
    c[i] = a[i] ^ b[i];
}
```

Nous avons dérouler la boucle en groupes de 32 itérations pour AVX2 et 64 pour AVX512.

4.1.2 Intel Intrinsics

Les instructions intrasèques pour le jeu d'instruction AVX2 ainsi que AVX512, permettent le traitement de plusieurs opérations en parallèle sur des vecteurs de 246bits (ymm) et 512bits pour avx512 (zmm).

Algorithme 4 : Fonction `mask_intrasic()`

```
#if __AVX512F__
void mask_intrasic(const u8 *restrict a, const u8 *restrict b, u8 *restrict c, u64 n)
{
  // a = load(a)
  // b = load(b)
  // x = xor(a,b)
  // c = store(x)
  for (u64 i = 0; i < n; i+=64)
  {
   __m512i va = _mm512_load_si512 (&a[i]);
   __m512i vb = _mm512_load_si512 (&b[i]);
   __m512i vx = _mm512_xor_si512 (va , vb);
    _mm512_store_si512 ( &c[i],vx);
  }
}
#else
void mask_intrasic(const u8 *restrict a, const u8 *restrict b, u8 *restrict c, u64 n)
{
  for (u64 i = 0; i < n; i+=32)
    {
      __m256i va = _mm256_load_si256 (&a[i]);
      __m256i vb = _mm256_load_si256 (&b[i]);
      __m256i vx = _mm256_xor_si256 (va,vb);
      _mm256_store_si256 (&c[i],vx);
    }
}
#endif
```

Le principal avantage de l'utilisation des intrasics est l'auto vectorisation du compilateur ainsi améliorer le parallélisme du code qui qui permet d'accéléré le traitement de mask.

4.1.3 OpenMP vectorization directive

Nous avons optimisé la fonction mask(), en utilisant les directives de compilation pour la vectorisation avec le code suivant :

Algorithme 5 : Fonction `mask_simd()`

```
void mask_simd(const u8 *restrict a, const u8 *restrict b, u8 *restrict c, u64 n)
{
  #pragma omp simd aligned(a,b,c)
  //#pragma unroll(32)
  for (u64 i = 0; i < n; i++)
    c[i] = a[i] ^ b[i];
}
```

Nous avont utilisé `#pragma omp simd` pour indiquer au compilateur que le block d'instruction qui suit peut être vectorisée, et cela à l'aide des instruction SIMD.Avec le mot clé `aligned` on informe le compilateur que les pointeur **a**, **b** et **c** sont alignés en mémoire cela peut simplifier la vectorisation.

4.2 Au niveau du code

4.2.1 Allignement de la mémoire

Au tout début le premier reflexe était d'aligner les allocation des données et mettre tous les pointeurs en `restrict`. L'allocation dynamique de la mémoire en **aligned_alloc** tout en spécifiant une limite d'alignement, signifie que la mémoire allouée sera alignée sur 64 octets, ce qui peut améliorer les performances des opérations SIMD.

Algorithme 5 : Allignement de la mémoire sur 64 octets

```
#define ALIGN 64
u8 *restrict cmp_mask = aligned_alloc(ALIGN ,sizeof(u8) * n);
```

4.2.2 Assembly

Nous avons essayé d'imiter le comportement du code en assembleur, en ajoutant un bloc `__asm__ volatile`, malheureusement nous n'avons pas pu fonctionner ce code.

Algorithme 6 : fonction mask_asm()

```
void mask_asm(const u8 *restrict a, const u8 *restrict b, u8 *restrict c, u64 n)
{

  __asm__ volatile(
        "xor %%rcx, %%rcx;\n"

        "loop:;\n"

        "vmovdqu (%[_a], %%rcx), %%ymm0;\n"
        "vmovdqu (%[_b], %%rcx), %%ymm1;\n"

        "vpxor %%ymm1, %%ymm0, %%ymm2;\n"

        "vmovdqu (%[_b], %%rcx), %%ymm2;\n"

        "add $32, %%rcx;\n"
        "cmp %[_n], %%rcx;\n"
        "jl loop;\n"

        : //outputs

        : //inputs
          [_a] "r" (a),
          [_b] "r" (b),
          [_c] "r" (c),
          [_n] "r" (n)

        : //clobber
          "cc", "memory", "rcx",
          "ymm0", "ymm1", "ymm2"
        );
}
```

Mais l'optimisation avec cette version peut être meilleure ou égalité au niveau de performance du code assembleur

de base généré lors de la compilation, car dans cet exemple de code nous n'avons pas à gérer plusieurs instructions et des jump (code simple avec des opérations de bases).

4.3 Parallélisme

Afin de mettre en œuvre le parallélisme les différentes version de la fonction `mask()` présentées ci-dessus sauf la version assembleur `mask_asm()`, car ce n'est pas possible à gérer le parallélisme dans les block *asm*,nous avons utilisé les directive de threading d'**OpenMp** fournies par la plupart des compilateurs modernes.

Algorithme 7 : fonction mask_asm_para()

```
1 void mask_para(const u8 *restrict a, const u8 *restrict b, u8 *restrict c, u64 n)
2 {
3   #pragma omp parallel for schedule(static)
4   for (u64 i = 0; i < n; i++)
5     c[i] = a[i] ^ b[i];
6 }
```

La directive `#pragma omp parallel for schedule(static)` indique au compilateur que la boucle doit être exécutée en parallèle sur plusieurs threads (à définir lors de l'execution du code). Le choix du scheduling static permet de partager la charge de travail de manière équitables entre tous les threads.

5 Performance profiling

Dans cette section nous visualisons les résultats de performance obtenus sur les différentes versions du code fourni, avec le compilateur **GCC** et **ICX** et avec différents nombre de threads pour les versions parallèle du code pour générer des séquences de longueur 20000000.

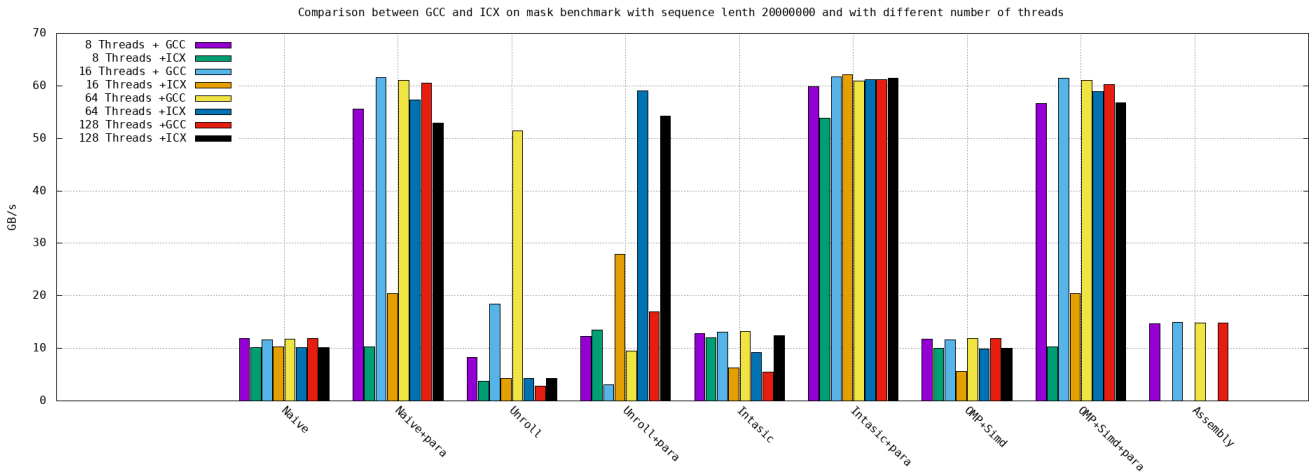

FIGURE 4 – Comparison between GCC and ICX on mask benchmark lenth 2000000 and with different number of threads

D'apres cette comparaison sur les performances exprimées en bande passante de mémoire en GiB/S (cf Figure 4), la plus efficaces pour les deux compilateurs GCC et ICX est la version mask_intrasic_para() et en deuxième place c'est la version parallélisé de mask_simd_para(). Pour ce cas d'exemple le compilateur GCC est souvent plus perfermant que ICX.

6 Conclusion

Dans ce TP, nous avons utilisé l'outil **MAQAO** sur le programme **mask** dans le but d'analyser ses performances et apporter les optimisations possibles. Le rapport d'analyse sur l'exécution du programme indiquait que plus de 99% de son temps d'exécution se faisait sur la fonction `mask()`.
Différentes implémentations pour cette fonction ont été pensées et testées pour couvrir les différents angles d'optimisations en commençant par les optimisations du compilateur avec les flags de compilation nécessaires. Ensuite, on a tenté des transformations du code afin d'améliorer la qualité du code généré par le compilateur. Pour avoir un bon déroulage de boucle et de la vectorisation pour bénéficier des instructions SIMD. Ensuite,nous avons fournis une version parallèle en utilisant les directives OpenMP. Enfin, nous avons fournis les benchmarks de toutes les versions de la fonction de base mask() et avec le compilateur GCC et ICX pour générer des séquences de longueur 2000000.