OBHPC TD2

C programming and performance metrics.

YU BOWEN

1. Constraints

1 – S'assurer que le laptop est connecté au secteur

2 – S'assurer que le CPU tourne a une frequence stable (cpupower)

La commande (cpupower frequency-set -f) peut être utilisée pour configurer le CPU afin qu'il fonctionne à une certaine fréquence dans la plupart des cas, cependant le pilote intel pstate fonctionnant dans l'un des modes actifs ne permet pas à l'utilisateur de sélectionner directement une fréquence spécifique.

Par conséquent, dans cette expérience, j'ai verrouillé la fréquence du CPU à la fréquence de fonctionnement la plus élevée qu'il supporte (cpupower frequency-set -g performance) plutôt que de l'ajuster dynamiquement..

```
yubowen@yubowen-VirtualBox:~/TD2$ cpupower
WARNING: cpupower not found for kernel 5.15.0-50

  You may need to install the following packages for this specific kernel:
    linux-tools-5.15.0-50-generic
    linux-cloud-tools-5.15.0-50-generic

  You may also want to install one of the following packages to keep up to date:
    linux-tools-generic
    linux-cloud-tools-generic
yubowen@yubowen-VirtualBox:~/TD2$ sudo apt install linux-tools-5.15.0-50-generic
```

Figure 1.1 set cpu

```
Setting up linux-tools-5.15.0-50-generic (5.15.0-50.56) ...
yubowen@yubowen-VirtualBox:~/TD2$ cpupower
Usage:  cpupower [-d|--debug] [-c|--cpu cpulist ] <command> [<args>]
Supported commands are:
        frequency-info
        frequency-set
        idle-info
        idle-set
        set
        info
        monitor
        help

Not all commands can make use of the -c cpulist option.

Use 'cpupower help <command>' for getting help for above commands.
```

Figure 1.2 set cpu

3 – Pinner le processus sur un coeur de calcul (taskset ou numactl)

Pendant l'expérience, j'ai utilisé (taskset -c 0 xxx) pour spécifier un cpu spécifique à exécuter.

2. Experiments

2.1 Extract target architecture information

2.1.1 CPU Information

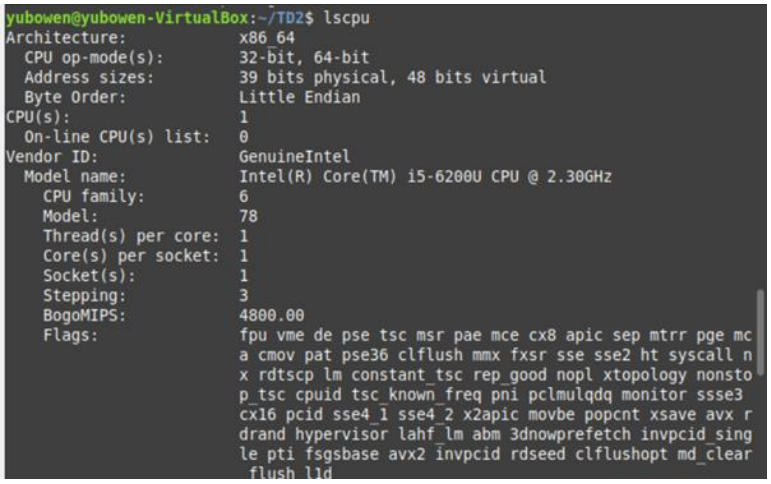


Figure 2.1 lscpu

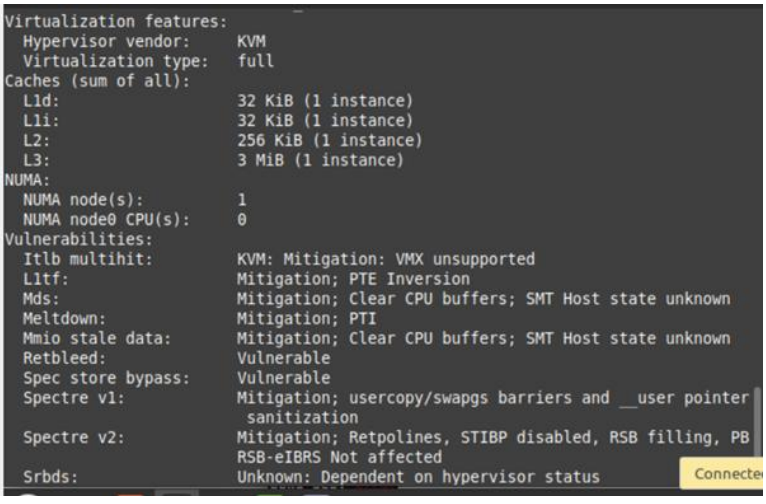


Figure 2.2 lscpu

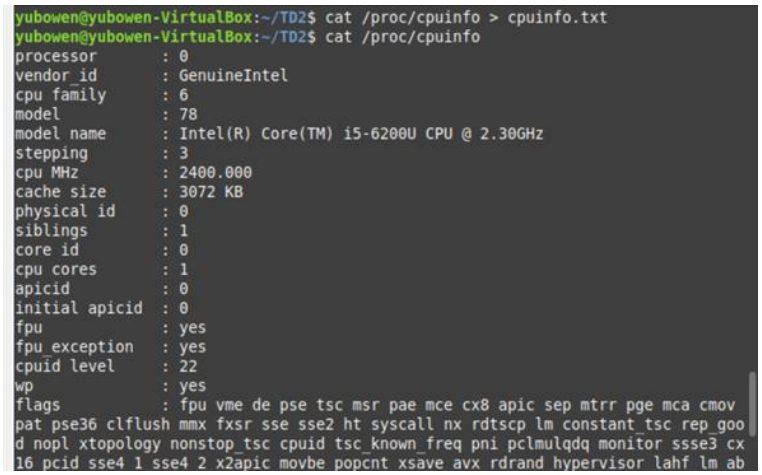


Figure 3 cat /proc/cpuinfo

2.1.2 Information about data caches

Consult the files in the following paths:

/sys/devices/system/cpu/cpu0/cache/index0/* for L1 cache

/sys/devices/system/cpu/cpu0/cache/index2/* for L2 cache

/sys/devices/system/cpu/cpu0/cache/index3/* for L3 cache

Take the L1 cache as example, I use the command:
tail /sys/devices/system/cpu/cpu0/cache/index0/* | cat

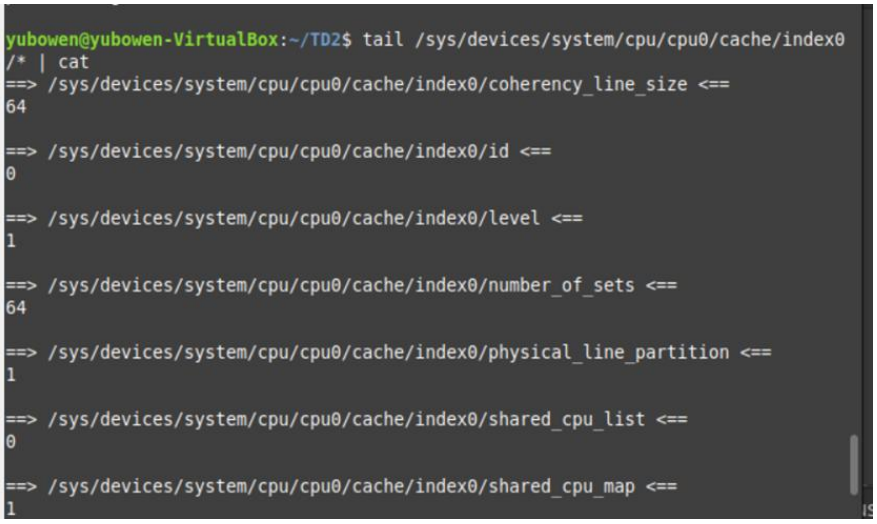


Figure 4 the L1 cache

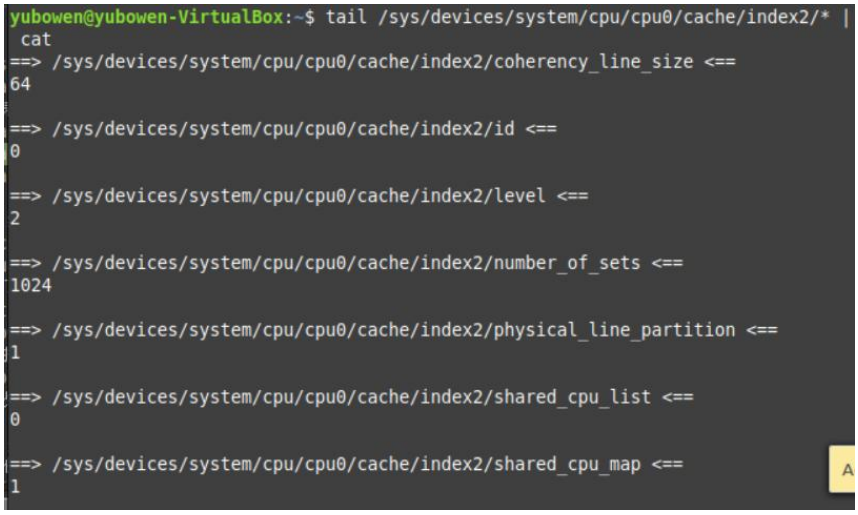


Figure 5 the L2 cache

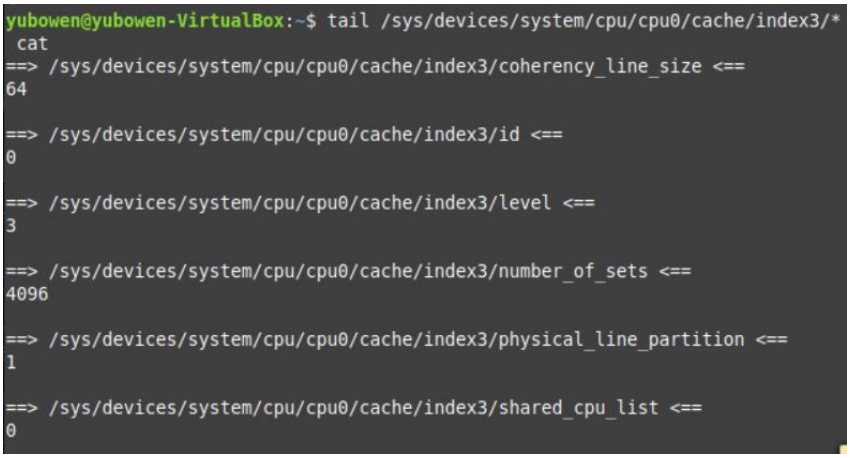


Figure 6 the L3 cache

2.2 Dgemm (N:128, R:100)

1 – Lancer le programme dgemm et récolter les mesures de performance pour chaque version dans un fichier à part (1 fichier par version).

2 – Modifier le Makefile fournit afin de tester plusieurs flags d'optimisation les compilateurs gcc, clang, icx, et icc (si installés).

J'ai examiné les performances de gcc, clang et icx sous quatre paramètres d'optimisation [-O0,-O1,-O2,-O3].

De plus, afin de permettre au script de s'exécuter plusieurs fois directement, j'ai ajouté la commande d'exécuter le clean en premier dans la commande all du Makefile.

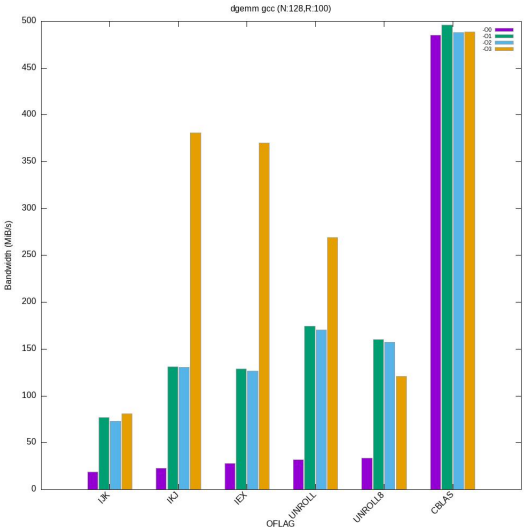
3 – Générer les fichiers de performance pour chacune des versions de la question 2.

4 – Rajouter une version de la fonction dgemm avec déroulage x8 et comparer ses performances aux autres versions.

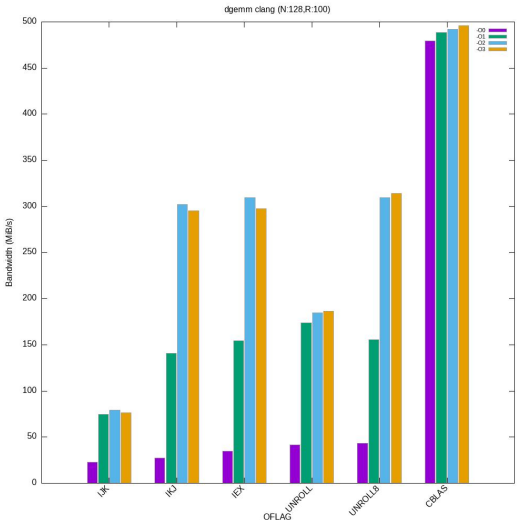
```c
void dgemm_unroll8(f64 *restrict a, f64 *restrict b, f64 *restrict c, u64 n)
{
#define UNROLL8 8
    for (u64 i = 0; i < n; i++)
    {
        for (u64 k = 0; k < n; k++)
        {
            const f64 _a_ = a[i * n + k];
            for (u64 j = 0; j < n; j += UNROLL8)
            {
                c[i * n + j]     +=  _a_ * b[k * n + j];
                c[i * n + j + 1] +=  _a_ * b[k * n + j + 1];
                c[i * n + j + 2] +=  _a_ * b[k * n + j + 2];
                c[i * n + j + 3] +=  _a_ * b[k * n + j + 3];
                c[i * n + j + 4] +=  _a_ * b[k * n + j + 4];
                c[i * n + j + 5] +=  _a_ * b[k * n + j + 5];
                c[i * n + j + 6] +=  _a_ * b[k * n + j + 6];
                c[i * n + j + 7] +=  _a_ * b[k * n + j + 7];
            }
        }
    }
}
```

Figure 7 code of unrolling x8

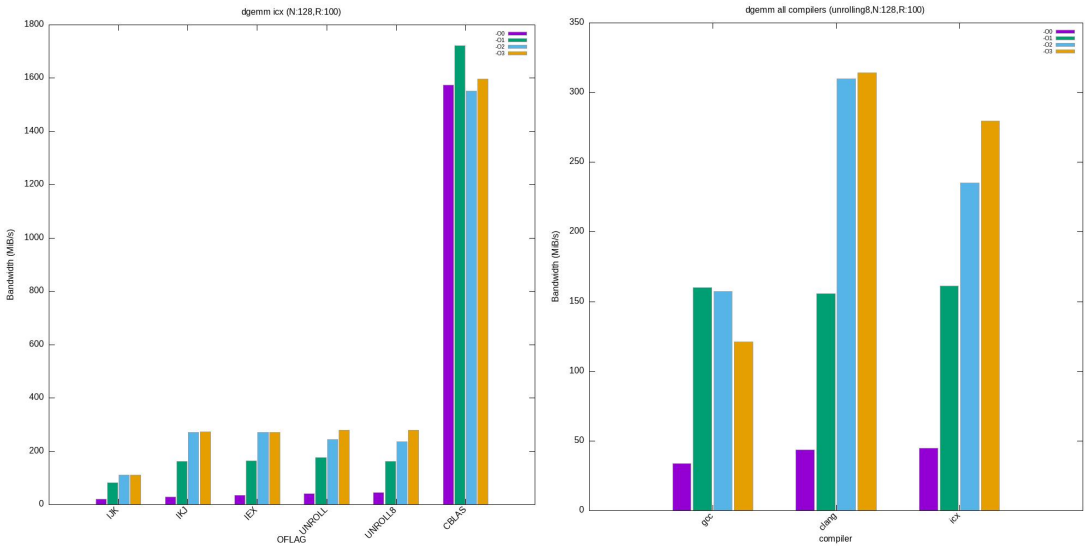
5 – Générer des graphiques (histogrammes) avec GNUPlot comparant les différentes versions pour chaque compilateur et un graphique comparant les version par compilateur.



(a)



(b)



(c) (d)

Figure 8 GNUplot of Dgemm

(a), (b) et (c) montrent la largeur de bande de différentes fonctions et différents paramètres d'optimisation avec les compilateurs gcc, clang et icx.

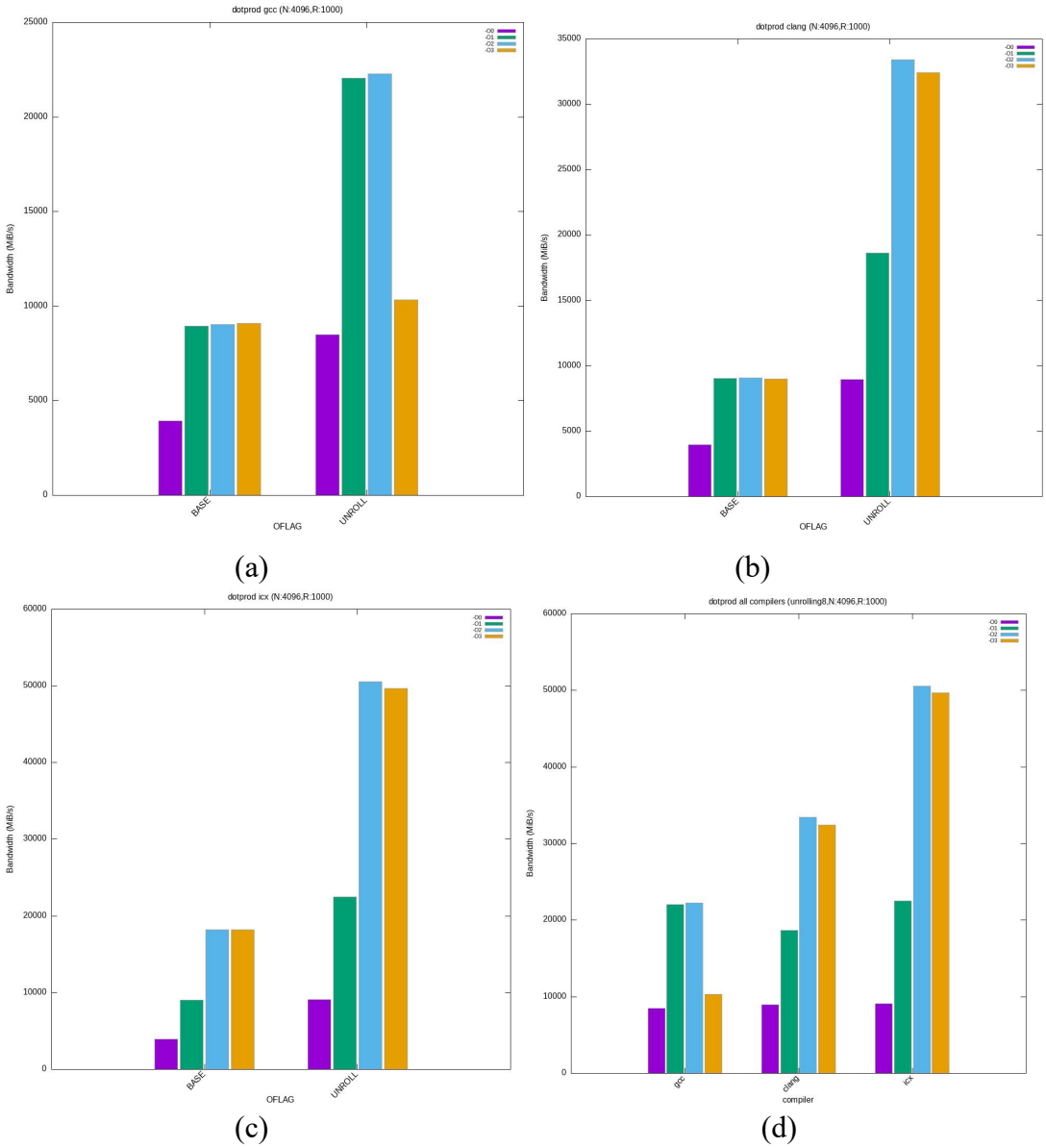
(d) montre la bande passante de différents compilateurs lors du déroulement de x8.

Comme on peut le voir dans la figure 8, dans la plupart des cas, l'augmentation des paramètres d'optimisation peut entraîner une plus grande augmentation de la vitesse du programme global.

Toutefois, il existe certaines exceptions ; par exemple, dans le cas du déroulement de la boucle x8, -O3 sera plus lent.

Dans une comparaison de compilateurs, gcc est le plus lent, clang est le plus rapide, et icx a une performance globale raisonnable.

2.3 Dotprod (N:4096, R:1000)



(a) (b)

(c) (d)

Figure 9 GNUplot of Dotprod

(a), (b) et (c) montrent la largeur de bande de différentes fonctions et différents paramètres d'optimisation avec les compilateurs gcc, clang et icx.
(d) montre la bande passante de différents compilateurs lors du déroulement de x8.

Comme on peut le voir sur la figure 9, dans la plupart des cas, des paramètres d'optimisation plus élevés peuvent apporter un plus grand degré d'amélioration de la vitesse à l'ensemble du programme.
-O3 et -O2 sont relativement proches.
Dans la comparaison entre les compilateurs, gcc est le plus lent, icx est le plus rapide, et clang a de bonnes performances générales.

2.4 Reduc (N:4096, R:1000)

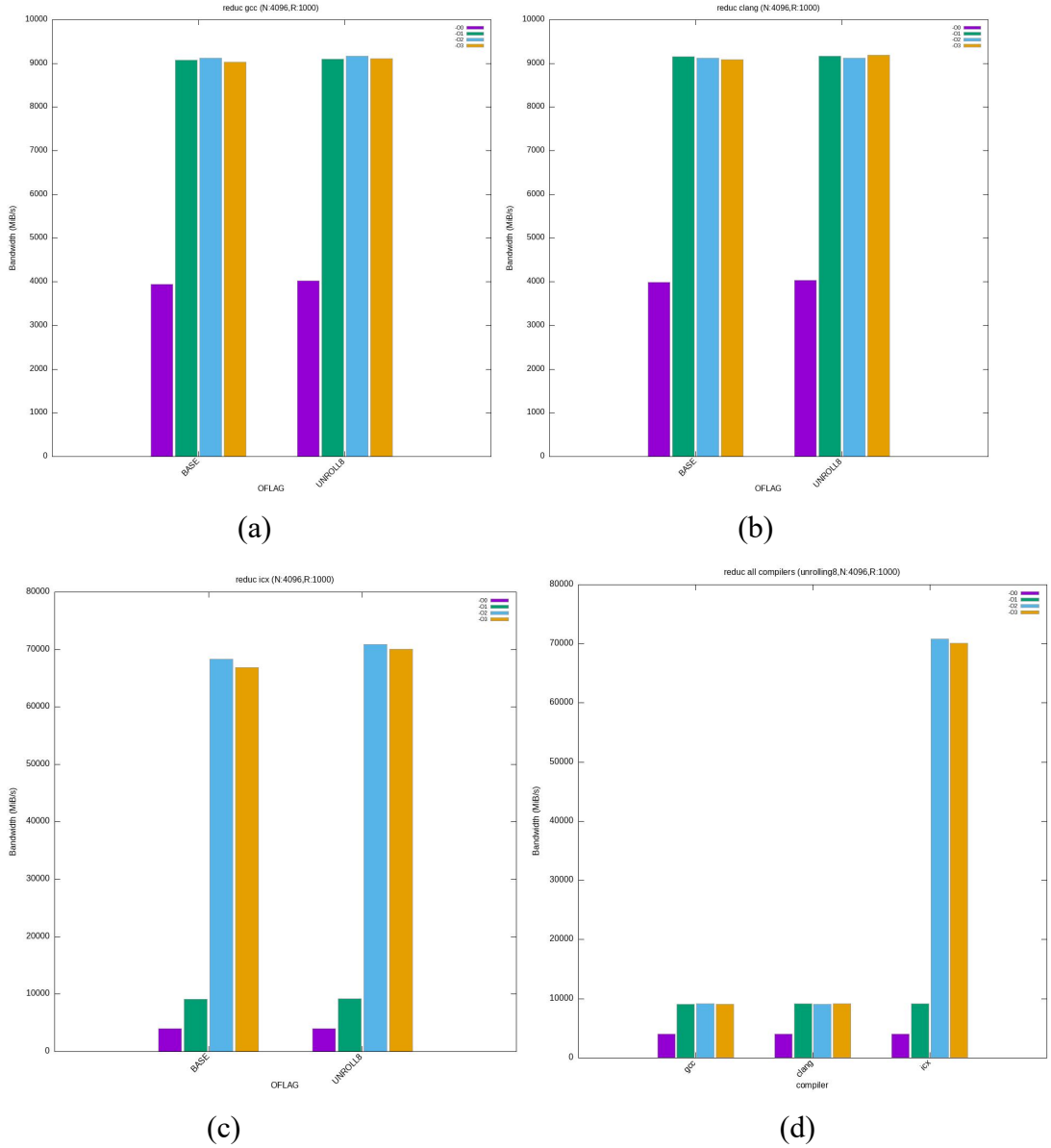


Figure 10 GNUplot of Reduc

(a), (b) et (c) montrent la largeur de bande de différentes fonctions et différents paramètres d'optimisation avec les compilateurs gcc, clang et icx.

(d) montre la bande passante de différents compilateurs lors du déroulement de x8.

Les résultats présentés à la figure 10 et à la figure 9 sont similaires. Dans la plupart des cas, des paramètres d'optimisation plus élevés peuvent apporter un plus grand degré d'amélioration de la vitesse à l'ensemble du programme.
-O3 et -O2 sont relativement proches.
Dans la comparaison entre les compilateurs, gcc est le plus lent, icx est le plus rapide et clang a de bonnes performances générales.