

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

```
yubowen@yubowen-VirtualBox:~/TD2$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Address sizes:          39 bits physical, 48 bits virtual
Byte Order:             Little Endian
CPU(s):                 1
On-line CPU(s) list:   0
Vendor ID:              GenuineIntel
Model name:             Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
CPU family:             6
Model:                  78
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):              1
Stepping:               3
BogoMIPS:               4800.00
Flags:                  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mc
a cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall n
x rdtscp lm constant tsc rep good nopl xtopology nonsto
p tsc cpuid tsc known freq pni pclmulqdq monitor ssse3
cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt xsave avx r
drand hypervisor lahf_lm abm 3dnowprefetch invpcid sing
le pti fsgsbase avx2 invpcid rdseed clflushopt md_clear
flush_lld
```

Figure 2.1 lscpu

```
Virtualization features:
Hypervisor vendor:     KVM
Virtualization type:   full
Caches (sum of all):
L1d:                   32 KiB (1 instance)
L1i:                   32 KiB (1 instance)
L2:                    256 KiB (1 instance)
L3:                    3 MiB (1 instance)
NUMA:
NUMA node(s):          1
NUMA node0 CPU(s):     0
Vulnerabilities:
Itlb multihit:         KVM: Mitigation: VMX unsupported
L1tf:                  Mitigation; PTE Inversion
Mds:                   Mitigation; Clear CPU buffers; SMT Host state unknown
Meltdown:              Mitigation; PTI
Mmio stale data:       Mitigation; Clear CPU buffers; SMT Host state unknown
Retbleed:              Vulnerable
Spec store bypass:     Vulnerable
Spectre v1:            Mitigation; usercopy/swapgs barriers and __user pointer
sanitization
Spectre v2:            Mitigation; Retpolines, STIBP disabled, RSB filling, PB
RSB-eIBRS Not affected
Srbds:                 Unknown: Dependent on hypervisor status
```

Figure 2.2 lscpu

```

yubowen@yubowen-VirtualBox:~/TD2$ cat /proc/cpuinfo > cpuinfo.txt
yubowen@yubowen-VirtualBox:~/TD2$ cat /proc/cpuinfo
processor       : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 78
model name    : Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
stepping      : 3
cpu MHz       : 2400.000
cache size    : 3072 KB
physical id   : 0
siblings      : 1
core id       : 0
cpu cores     : 1
apicid        : 0
initial apicid : 0
fpu           : yes
fpu_exception : yes
cpuid level   : 22
wp            : yes
flags         : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_goo
d nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq monitor ssse3 cx
16 pcid sse4_1 sse4_2 x2apic movbe popcnt xsave avx rdrand hypervisor lahf_lm ab

```

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`

```

yubowen@yubowen-VirtualBox:~/TD2$ tail /sys/devices/system/cpu/cpu0/cache/index0
/* | cat
==> /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size <==
64

==> /sys/devices/system/cpu/cpu0/cache/index0/id <==
0

==> /sys/devices/system/cpu/cpu0/cache/index0/level <==
1

==> /sys/devices/system/cpu/cpu0/cache/index0/number_of_sets <==
64

==> /sys/devices/system/cpu/cpu0/cache/index0/physical_line_partition <==
1

==> /sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list <==
0

==> /sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_map <==
1

```

Figure 4 the L1 cache

```

yubowen@yubowen-VirtualBox:~$ tail /sys/devices/system/cpu/cpu0/cache/index2/* |
cat
==> /sys/devices/system/cpu/cpu0/cache/index2/coherency_line_size <==
64

==> /sys/devices/system/cpu/cpu0/cache/index2/id <==
0

==> /sys/devices/system/cpu/cpu0/cache/index2/level <==
2

==> /sys/devices/system/cpu/cpu0/cache/index2/number_of_sets <==
1024

==> /sys/devices/system/cpu/cpu0/cache/index2/physical_line_partition <==
1

==> /sys/devices/system/cpu/cpu0/cache/index2/shared_cpu_list <==
0

==> /sys/devices/system/cpu/cpu0/cache/index2/shared_cpu_map <==
1

```

Figure 5 the L2 cache

```

yubowen@yubowen-VirtualBox:~$ tail /sys/devices/system/cpu/cpu0/cache/index3/*
cat
==> /sys/devices/system/cpu/cpu0/cache/index3/coherency_line_size <==
64

==> /sys/devices/system/cpu/cpu0/cache/index3/id <==
0

==> /sys/devices/system/cpu/cpu0/cache/index3/level <==
3

==> /sys/devices/system/cpu/cpu0/cache/index3/number_of_sets <==
4096

==> /sys/devices/system/cpu/cpu0/cache/index3/physical_line_partition <==
1

==> /sys/devices/system/cpu/cpu0/cache/index3/shared_cpu_list <==
0

```

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.

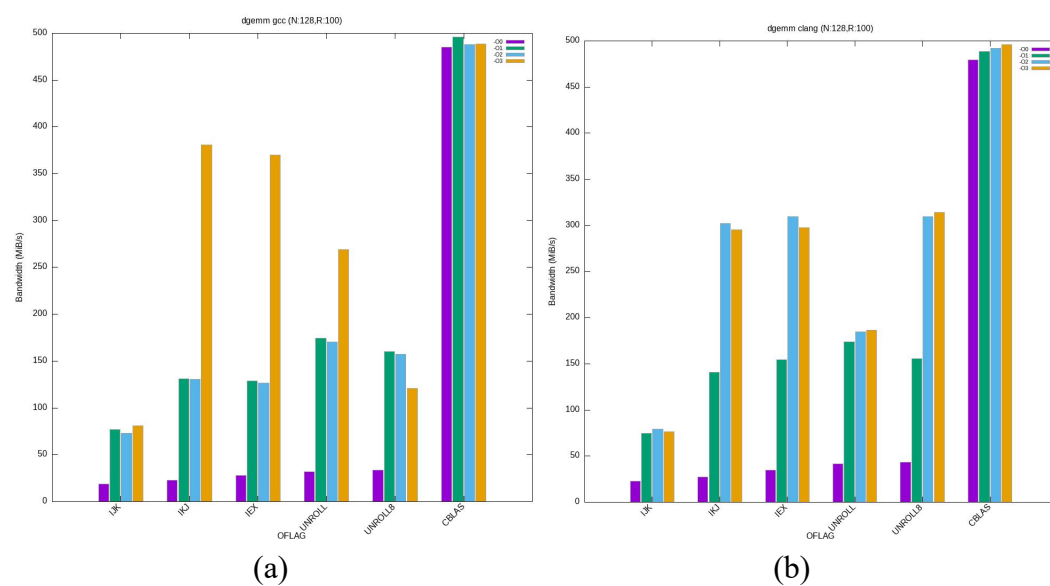
```

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.



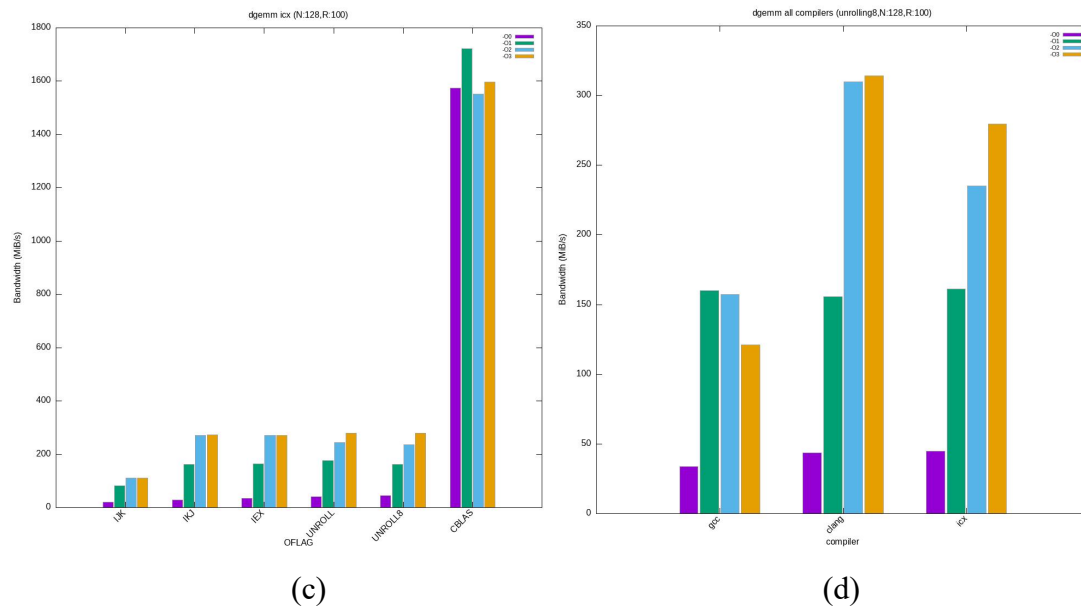


Figure 8 GNUplot of Dgemm

(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)

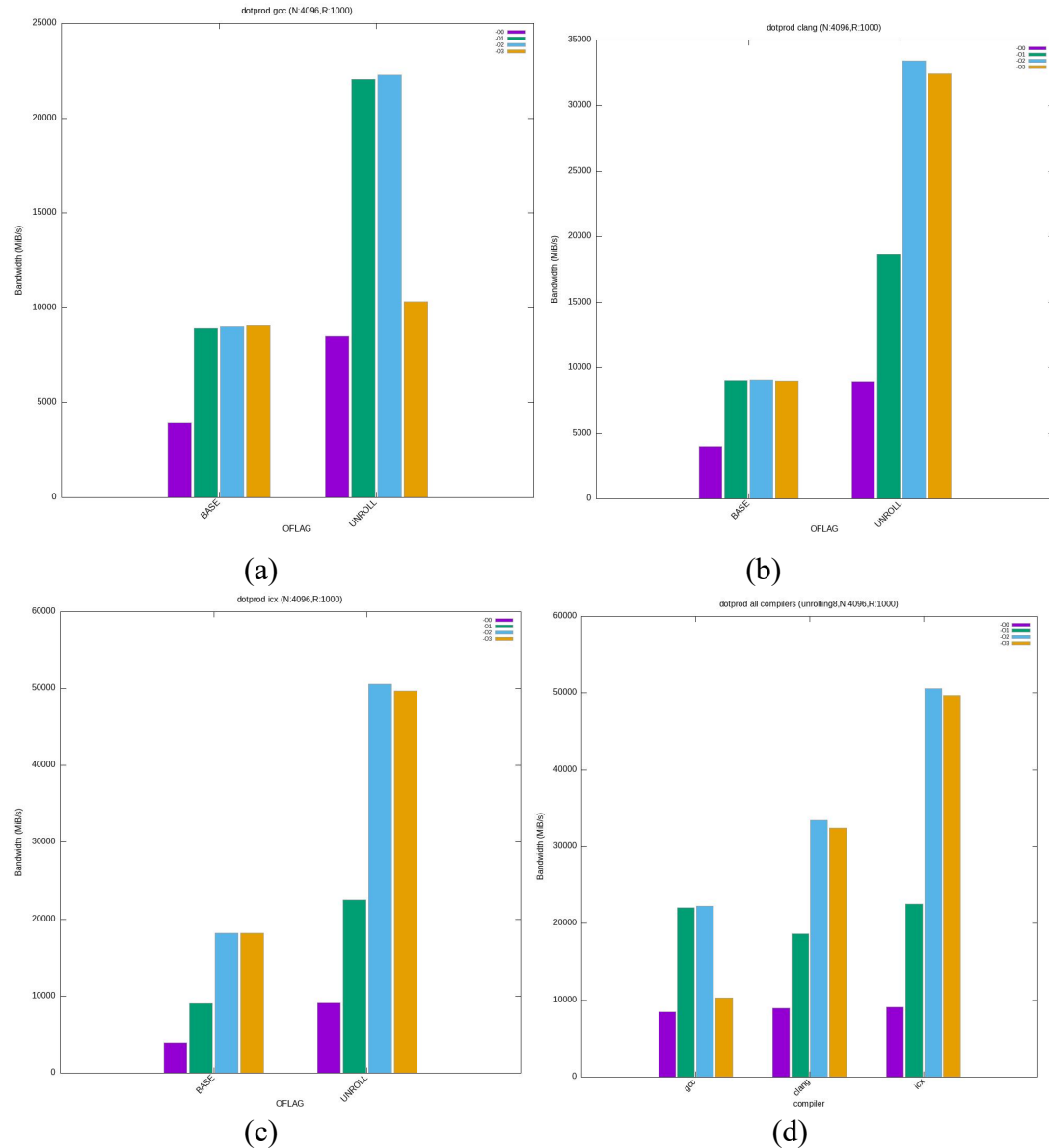


Figure 9 GNUpot of Dotprod

(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)

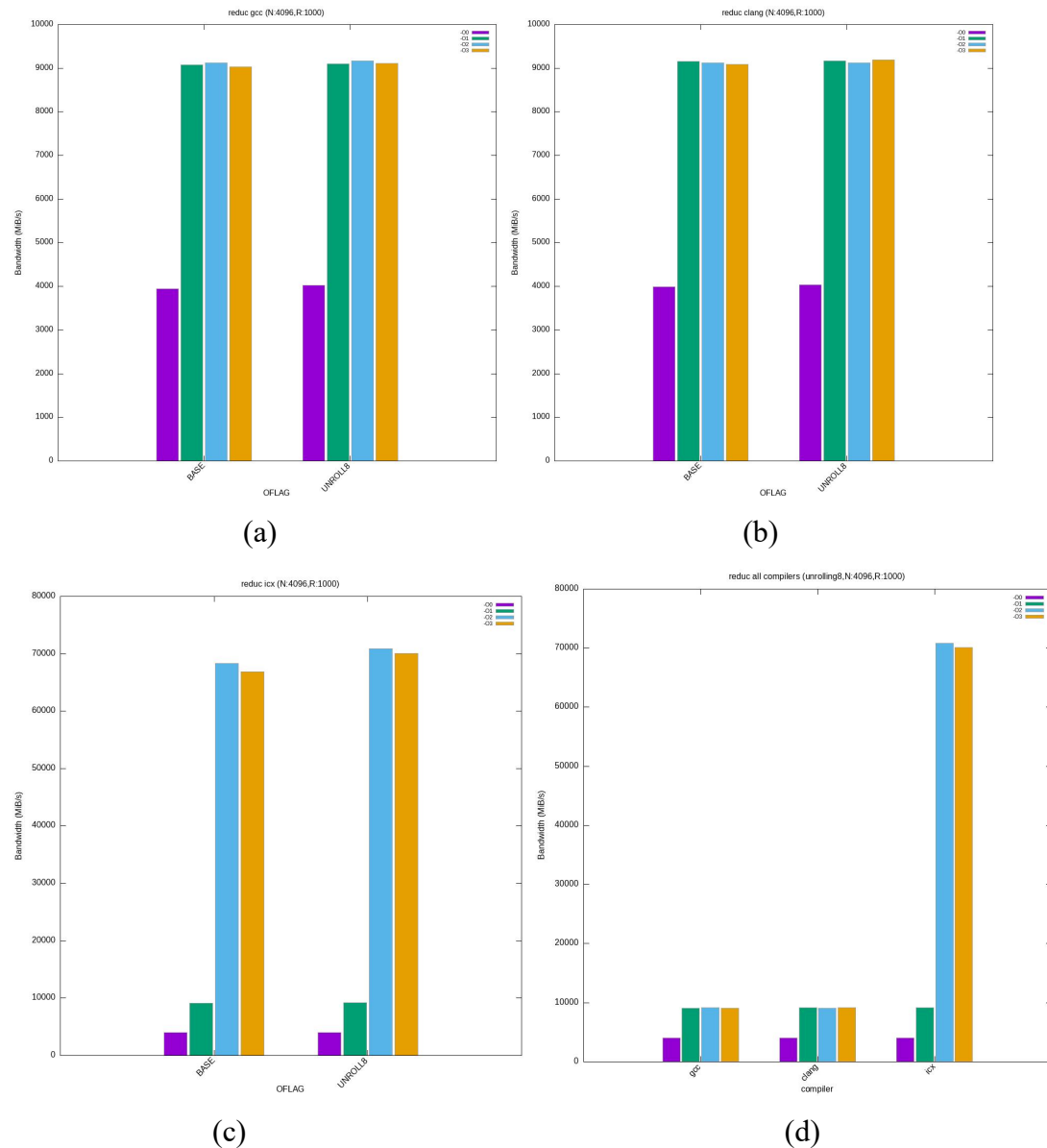


Figure 10 GNUplot of Reduc

(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.