Rapport de TP2 OBCHPS

Yutai ZHAO

I. Introduction

Dans ce rapport, nous comparons les résultats obtenus en exécutant les programmes *dotprod* et *réduc*. Le programme ***dotprod*** calcule le produit scalaire de 2 vecteurs générés aléatoirement et ***réduc*** calcule la somme de toutes les valeurs du vecteur. Les calculs de chaque programme sont réalisés par 2 algorithmes, une version base et une version déroulée 8 fois. Les comparaisons porteront sur principalement 2 informations : l'écart type de latences (*stddev*) et la vitesse à laquelle les calculs sont faits (*mbps*), pour la suite nous utiliserons *stddev* et *mbps* comme notation pour parler de ces 2 informations. De plus, lors de l'analyse de *mbps*, le mot performance est interchangeable avec *mbps*. Les descriptions et comparaisons seront fournies sous formes de tableaux, de graphes. Des résumés descriptifs textuels seront également fournis, mais les **conclusions** sont plus intéressantes à lire.

II. Table de matières

III. Implémentation de 2 fonctions

➢ ***dotprod*** **version déroulée 8 fois :**

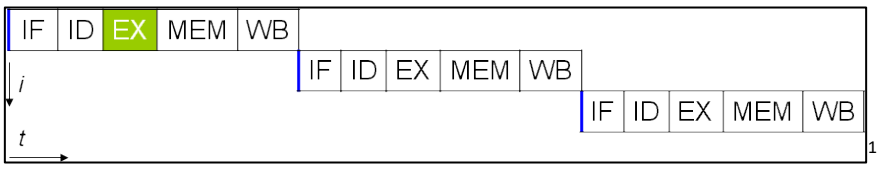
```
f64 dotprod_unroll(f64 *restrict a, f64 *restrict b, u64 n)
{
    double d = 0 ;
    double r0 = 0 ;
    double r1 = 0 ;
    double r2 = 0 ;
    double r3 = 0 ;
    double r4 = 0 ;
    double r5 = 0 ;
    double r6 = 0 ;
    double r7 = 0 ;
    for (u64 i = 0; i < n; i+=8){
        r0 += a[i] * b[i];
        r1 += a[i+1] * b[i+1];
        r2 += a[i+2] * b[i+2];
        r3 += a[i+3] * b[i+3];
        r4 += a[i+4] * b[i+4];
        r5 += a[i+5] * b[i+5];
        r6 += a[i+6] * b[i+6];
        r7 += a[i+7] * b[i+7];
    }
    d = r0+r1+r2+r3+r4+r5+r6+r7;
  return d;
}
```

➢ ***reduc*** **version déroulée 8 fois :**

```
f64 reduc_unroll(f64 *restrict a, u64 n) \\check kernels.c to see another version
{
  double d = 0.0
    for (u64 i = 0; i < n; i+=8){
        d += a[i] + a[i+1] + a[i+2] + a[i+3] + a[i+4] + a[i+5] + a[i+6] + a[i+7];
    }
  return d;
}
```

IV. les versions optimisées à priori : versions déroulées

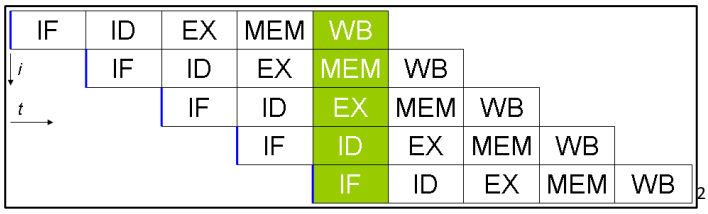
Les instructions sans pipeline sont exécutées l'une après l'autre



[1]

[1] https://fr.wikipedia.org/wiki/Pipeline_(architecture_des_processeurs)

Le pipeline permet à une instruction de s'exécuter sans attendre que la précédente soit terminée :



[2]

Donc, à travers les fonctions implémentées, qui sont les versions déroulées x8, on essaye de remplir le pipeline afin d'exécuter 8 instructions parallèlement.

V. Environnement et compilation

Chaque programme, à savoir *dotprod* et *reduc*, est exécuté sur un MacOS, les informations sur le CPU, le CACHE sont fournies dans *cpu_info.txt, cache_info.txt*. Vu que les commandes données ne fonctionnent pas sur mon OS, les informations sont obtenues respectivement avec ses deux commandes : `sysctl -a | grep machdep.cpu` et `sysctl -a | grep cache`. Je n'ai pas trouvé de commandes équivalentes pour `cpupower` et `taskset`.
Chaque programme est généré par 2 compilateurs GCC et CLANG dont les versions sont respectivement obtenues par `brew info gcc` et `clang --version` sont fournies dans un *compiler_info.txt* : la version de CLANG est 14.0.6, la version de GCC est 13.2.0.
Chaque programme est généré par des flags d'optimisation différents (-O0,-O1,-O2,-O3,-Ofast). Le choix des flags est premièrement dû au fait que ce sont des flags présentés dans le cours, sinon nous pouvons également consulter le site de GCC pour d'autres flags comme -Os,-Oz, mais comme ils *« optimize for size rather than speed […] enabling most -O2 optimizations »*[3] nous nous n'intéresserons pas aux ces flags.

VI. Choix de paramètres et Exécution

Pour éviter de tomber sur une valeur aberrante, qui a eu lieu souvent lors de la 1ere exécution, chaque programme est exécuté 101 fois et **il n'y que la médiane sera prise en compte pour la description et la comparaison de mbps**.
Chaque programme est exécuté avec les paramètres n=8 r=1000. Vu que la fonction « *aligned_alloc* » dans le fichier *main.c* prend « *ALIGN64* » et « *size* » comme arguments et que « 64-bit aligned is 8 bytes aligned »[4]. Ainsi, « *size* » qui est *« the number of bytes to allocate. An integral multiple of alignment »*[5] oblige que n prenne une valeur étant un multiple de 8. J'ai hésité pour la valeur de n en voulant tester les versions pour n = 8000, r=10 qui fait que le cache 1 soit bien rempli, mais les valeurs de *stddev* sont tous devenues très grandes. Cela signifie que les valeurs de *mbps* sont très diverses, donc elles deviennent insignifiantes.

VII. Flags d'optimisation

Les comportements des flags d'optimisation dépendent de compilateurs ainsi que leurs versions, mais en général les optimisations sont de plus en plus agressives. Voici une brève description prise pendant le Tp, d'ailleurs je n'ai pas fait attention de noter le compilateur utilisé ni sa version :
-O0 : interdit toute optimisation. -O1 : accès aux certaines optimisations, il permet par exemple au compilateur de supprimer les variables jamais utilisées. -O2 : accès aux optimisations de -O1 et d'autres, il permet par exemple au compilateur de remplacer un appel d'une fonction par le corps de la fonction elle-même afin d'éviter de faire un appel.

---

[2] https://fr.wikipedia.org/wiki/Pipeline_(architecture_des_processeurs)
[3] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[4] https://en.wikipedia.org/wiki/Data_structure_alignment
[5] https://en.cppreference.com/w/c/memory/aligned_alloc

-O3 : accès aux optimisations de -O2 et d'autres. -Ofast : accès aux optimisations de -O3 et d'autres, il permet au compilateur de nuire notamment les précisions des calculs maths pour accélérer un programme.

Nous pouvons d'ailleurs jeter un coup d'œil sur la version BASE de *dotprod* en langage assembleur provenant du site COMPILER EXPLORER[6] afin d'étudier plus précisément les effets des flags d'optimisation sur le code. Les paramètres de compilation tels que le compilateur choisi et sa version sont bien ajustés et correspondent aux informations fournies, c'est-à-dire que : CLANG : v.14.0.0, GCC : v.13.2.0.

Dotprod – BASE – GCC – O0 :

```
dotprod_base:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rdi, -24(%rbp)
        movq    %rsi, -32(%rbp)
        movq    %rdx, -40(%rbp)
        pxor    %xmm0, %xmm0
        movsd   %xmm0, -8(%rbp)
        movq    $0, -16(%rbp)
        jmp     .L2
.L3:
        movq    -16(%rbp), %rax
        leaq    0(,%rax,8), %rdx
        movq    -24(%rbp), %rax
        addq    %rdx, %rax
        movsd   (%rax), %xmm1
        movq    -16(%rbp), %rax
        leaq    0(,%rax,8), %rdx
        movq    -32(%rbp), %rax
        addq    %rdx, %rax
        movsd   (%rax), %xmm0
        mulsd   %xmm1, %xmm0
        movsd   -8(%rbp), %xmm1
        addsd   %xmm1, %xmm0
        movsd   %xmm0, -8(%rbp)
        addq    $1, -16(%rbp)
.L2:
        movq    -16(%rbp), %rax
        cmpq    -40(%rbp), %rax
        jb      .L3
        movsd   -8(%rbp), %xmm0
        movq    %xmm0, %rax
        movq    %rax, %xmm0
        popq    %rbp
        ret
```

Dotprod – BASE – GCC – O1 :

```
dotprod_base:
        testq   %rdx, %rdx
        je      .L4
        movl    $0, %eax
        pxor    %xmm1, %xmm1
.L3:
        movsd   (%rdi,%rax,8), %xmm0
        mulsd   (%rsi,%rax,8), %xmm0
        addsd   %xmm0, %xmm1
        addq    $1, %rax
        cmpq    %rax, %rdx
        jne     .L3
.L1:
        movapd  %xmm1, %xmm0
        ret
.L4:
        pxor    %xmm1, %xmm1
        jmp     .L1
```

Dotprod – BASE – GCC – O2 :

```
dotprod_base:
        testq   %rdx, %rdx
        je      .L4
        xorl    %eax, %eax
        pxor    %xmm1, %xmm1
.L3:
        movsd   (%rdi,%rax,8), %xmm0
        mulsd   (%rsi,%rax,8), %xmm0
        addq    $1, %rax
        addsd   %xmm0, %xmm1
        cmpq    %rax, %rdx
        jne     .L3
        movapd  %xmm1, %xmm0
        ret
.L4:
        pxor    %xmm1, %xmm1
        movapd  %xmm1, %xmm0
        ret
```

[6] https://godbolt.org

Dotprod – BASE – GCC – O3 :

```
dotprod_base:
        testq   %rdx, %rdx
        je      .L7
        cmpq    $1, %rdx
        je      .L8
        movq    %rdx, %rcx
        xorl    %eax, %eax
        pxor    %xmm0, %xmm0
        shrq    %rcx
        salq    $4, %rcx
.L4:
        movupd  (%rdi,%rax), %xmm1
        movupd  (%rsi,%rax), %xmm3
        addq    $16, %rax
        mulpd   %xmm3, %.xmm1
        addsd   %xmm1, %xmm0
        unpckhpd        %xmm1, %xmm1
        addsd   %xmm1, %xmm0
        cmpq    %rcx, %rax
        jne     .L4
        testb   $1, %dl
        je      .L1
        andq    $-2, %rdx
.L3:
        movsd   (%rsi,%rdx,8), %xmm1
        mulsd   (%rdi,%rdx,8), %xmm1
        addsd   %xmm1, %xmm0
        ret
.L7:
        pxor    %xmm0, %xmm0
.L1:
        ret
.L8:
        xorl    %edx, %edx
        pxor    %xmm0, %xmm0
        jmp     .L3
```

Dotprod – BASE – GCC – Ofast :

```
dotprod_base:
        testq   %rdx, %rdx
        je      .L7
        cmpq    $1, %rdx
        je      .L8
        movq    %rdx, %rcx
        xorl    %eax, %eax
        pxor    %xmm2, %xmm2
        shrq    %rcx
        salq    $4, %rcx
.L4:
        movupd  (%rdi,%rax), %xmm0
        movupd  (%rsi,%rax), %xmm3
        addq    $16, %rax
        mulpd   %xmm3, %xmm0
        addpd   %xmm0, %xmm2
        cmpq    %rcx, %rax
        jne     .L4
        movapd  %xmm2, %xmm1
        unpckhpd        %xmm2, %xmm1
        addpd   %xmm2, %xmm1
        testb   $1, %dl
        je      .L1
        andq    $-2, %rdx
.L3:
        movsd   (%rsi,%rdx,8), %xmm0
        mulsd   (%rdi,%rdx,8), %xmm0
        addsd   %xmm0, %xmm1
.L1:
        movapd  %xmm1, %xmm0
        ret
.L7:
        pxor    %xmm1, %xmm1
        movapd  %xmm1, %xmm0
        ret
.L8:
        xorl    %edx, %edx
        pxor    %xmm1, %xmm1
        jmp     .L3
```

Dotprod – BASE – CLANG – O0 :

```
dotprod_base:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rdi, -8(%rbp)
        movq    %rsi, -16(%rbp)
        movq    %rdx, -24(%rbp)
        xorps   %xmm0, %xmm0
        movsd   %xmm0, -32(%rbp)
        movq    $0, -40(%rbp)
.LBB0_1:
        movq    -40(%rbp), %rax
        cmpq    -24(%rbp), %rax
        jae     .LBB0_4
        movq    -8(%rbp), %rax
        movq    -40(%rbp), %rcx
        movsd   (%rax,%rcx,8), %xmm0
        movq    -16(%rbp), %rax
        movq    -40(%rbp), %rcx
        movsd   (%rax,%rcx,8), %xmm2
        movsd   -32(%rbp), %xmm1
        mulsd   %xmm2, %xmm0
        addsd   %xmm1, %xmm0
        movsd   %xmm0, -32(%rbp)
        movq    -40(%rbp), %rax
        addq    $1, %rax
        movq    %rax, -40(%rbp)
        jmp     .LBB0_1
.LBB0_4:
        movsd   -32(%rbp), %xmm0
        popq    %rbp
        retq
```

Dotprod – BASE – CLANG – O1 :

```
dotprod_base:
        xorpd   %xmm0, %xmm0
        testq   %rdx, %rdx
        je      .LBB0_3
        xorl    %eax, %eax
.LBB0_2:
        movsd   (%rdi,%rax,8), %xmm1
        mulsd   (%rsi,%rax,8), %xmm1
        addsd   %xmm1, %xmm0
        addq    $1, %rax
        cmpq    %rax, %rdx
        jne     .LBB0_2
.LBB0_3:
        retq
```

Dotprod – BASE – CLANG – O2 -O3 :

```
dotprod_base:
        testq   %rdx, %rdx
        je      .LBB0_1
        leaq    -1(%rdx), %rcx
        movl    %edx, %eax
        andl    $3, %eax
        cmpq    $3, %rcx
        jae     .LBB0_8
        xorpd   %xmm0, %xmm0
        xorl    %ecx, %ecx
        jmp     .LBB0_4
.LBB0_1:
        xorps   %xmm0, %xmm0
        retq
.LBB0_8:
        andq    $-4, %rdx
        xorpd   %xmm0, %xmm0
        xorl    %ecx, %ecx
.LBB0_9:
        movsd   (%rdi,%rcx,8), %xmm1
        movsd   8(%rdi,%rcx,8), %xmm2
        mulsd   (%rsi,%rcx,8), %xmm1
        mulsd   8(%rsi,%rcx,8), %xmm2
        addsd   %xmm0, %xmm1
        movsd   16(%rdi,%rcx,8), %xmm3
        mulsd   16(%rsi,%rcx,8), %xmm3
        addsd   %xmm1, %xmm2
        movsd   24(%rdi,%rcx,8), %xmm0
        mulsd   24(%rsi,%rcx,8), %xmm0
        addsd   %xmm2, %xmm3
        addsd   %xmm3, %xmm0
        addq    $4, %rcx
        cmpq    %rcx, %rdx
        jne     .LBB0_9
.LBB0_4:
        testq   %rax, %rax
        je      .LBB0_7
        leaq    (%rsi,%rcx,8), %rdx
        leaq    (%rdi,%rcx,8), %rcx
        xorl    %esi, %esi
.LBB0_6:
        movsd   (%rcx,%rsi,8), %xmm1
        mulsd   (%rdx,%rsi,8), %xmm1
        addsd   %xmm1, %xmm0
        addq    $1, %rsi
        cmpq    %rsi, %rax
        jne     .LBB0_6
.LBB0_7:
        retq
```

Dotprod – BASE – CLANG – Ofast :

```
dotprod_base:
        testq   %rdx, %rdx
        je      .LBB0_1
        cmpq    $4, %rdx
        jae     .LBB0_4
        xorpd   %xmm0, %xmm0
        xorl    %eax, %eax
        jmp     .LBB0_11
.LBB0_1:
        xorps   %xmm0, %xmm0
        retq
.LBB0_4:
        movq    %rdx, %rax
        andq    $-4, %rax
        leaq    -4(%rax), %rcx
        movq    %rcx, %r8
        shrq    $2, %r8
        addq    $1, %r8
        testq   %rcx, %rcx
        je      .LBB0_5
        movq    %r8, %r9
        andq    $-2, %r9
        xorpd   %xmm1, %xmm1
        xorl    %ecx, %ecx
        xorpd   %xmm0, %xmm0
.LBB0_7:
        movupd  (%rdi,%rcx,8), %xmm2
        movupd  16(%rdi,%rcx,8), %xmm3
        movupd  32(%rdi,%rcx,8), %xmm4
        movupd  48(%rdi,%rcx,8), %xmm5
        movupd  (%rsi,%rcx,8), %xmm6
        mulpd   %xmm2, %xmm6
        addpd   %xmm1, %xmm6
        movupd  16(%rsi,%rcx,8), %xmm2
        mulpd   %xmm3, %xmm2
        addpd   %xmm0, %xmm2
        movupd  32(%rsi,%rcx,8), %xmm1
        mulpd   %xmm4, %xmm1
        addpd   %xmm6, %xmm1
        movupd  48(%rsi,%rcx,8), %xmm0
        mulpd   %xmm5, %xmm0
        addpd   %xmm2, %xmm0
        addq    $8, %rcx
        addq    $-2, %r9
        jne     .LBB0_7
        testb   $1, %r8b
        je      .LBB0_10
.LBB0_9:
        movupd  (%rdi,%rcx,8), %xmm2
        movupd  16(%rdi,%rcx,8), %xmm3
        movupd  (%rsi,%rcx,8), %xmm4
        mulpd   %xmm2, %xmm4
        addpd   %xmm4, %xmm1
        movupd  16(%rsi,%rcx,8), %xmm2
        mulpd   %xmm3, %xmm2
        addpd   %xmm2, %xmm0
.LBB0_10:
        addpd   %xmm0, %xmm1
        movapd  %xmm1, %xmm0
        unpckhpd        %xmm1, %xmm0
        addsd   %xmm1, %xmm0
        cmpq    %rdx, %rax
        je      .LBB0_12
.LBB0_11:
        movsd   (%rsi,%rax,8), %xmm1
        mulsd   (%rdi,%rax,8), %xmm1
        addsd   %xmm1, %xmm0
        addq    $1, %rax
        cmpq    %rax, %rdx
        jne     .LBB0_11
.LBB0_12:
        retq
.LBB0_5:
        xorpd   %xmm1, %xmm1
        xorl    %ecx, %ecx
        xorpd   %xmm0, %xmm0
        testb   $1, %r8b
        jne     .LBB0_9
        jmp     .LBB0_10
```

D'après ces lignes surlignées en jaune, nous pouvons en tirer quelques remarques :

Quelques optimisations de flags pour dotprod BASE								
	GCC O1	GCC O2	GCC O3	GCC Of	CLANG O1	CLANG O2	CLANG O3	CLANG Of
UNROLL 4						✓	✓	✓
SD	✓	✓			✓	✓	✓	
PD			✓	✓				✓

UNROLL 4 : Déroulage x4, *SD* : scalaire double, *PD* : packed double (vectorisation de deux SD)

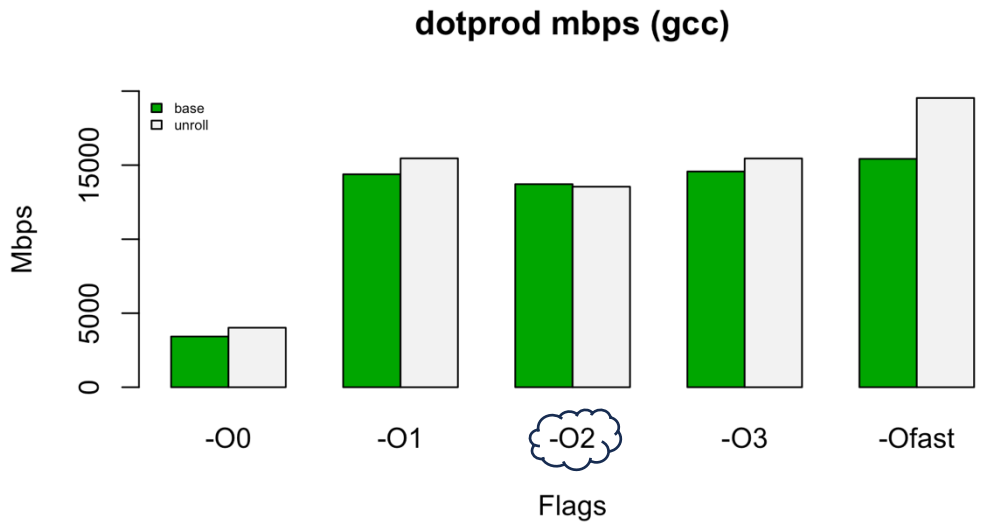
VIII. Description et Comparaison des résultats

➢ Résultats de dotprod

❖ Comparaison de mbps (Mib/s) entre version BASE et UNROLL

La performance de chaque version est fortement influencée par les flags d'optimisation. En effet, nous pouvons même dire que, en compilant avec flags différents, de nombreuses versions dérivées sont générées. Donc en fin, nous n'avons pas que 2 versions, mais environs 2x5 versions. Donc, nous pouvons prétendre qu'une comparaison suivant ces étapes pourrait faciliter la compréhension et la lecture des résultats : d'abord comparer l'évolution des *mbps* d'une version compilée avec flags différents, puis comparer les *mbps* entre différentes versions compilées avec le même flag.

GCC	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O0	BASE	0.125	0	0	8	1000	277.812	17.688	18.061	17.819	0.074(0.418%)	3425.279
O0	UNROLL	0.125	0	0	8	1000	39.193	15.128	15.31	15.182	0.034(0.223%)	4020.103
GCC	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O1	BASE	0.125	0	0	8	1000	106.096	4.226	4.325	4.243	0.017(0.399%)	14383.468
O1	UNROLL	0.125	0	0	8	1000	533.195	3.936	3.986	3.948	0.009(0.230%)	15458.817
GCC	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O2	BASE	0.125	0	0	8	1000	166.315	3.938	4.517	4.45	0.152(3.412%)	13715.765
O2	UNROLL	0.125	0	0	8	1000	176	4.495	4.539	4.506	0.009(0.202%)	13544.579
GCC	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O3	BASE	0.125	0	0	8	1000	126.593	3.669	4.234	4.189	0.121(2.877%)	14570.762
O3	UNROLL	0.125	0	0	8	1000	122.081	3.939	3.995	3.95	0.010(0.261%)	15451.464
GCC	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
Ofast	BASE	0.125	0	0	8	1000	187.507	3.938	4.265	3.957	0.056(1.404%)	15423.068
Ofast	UNROLL	0.125	0	0	8	1000	743.226	3.107	3.383	3.124	0.048(1.526%)	19535.228

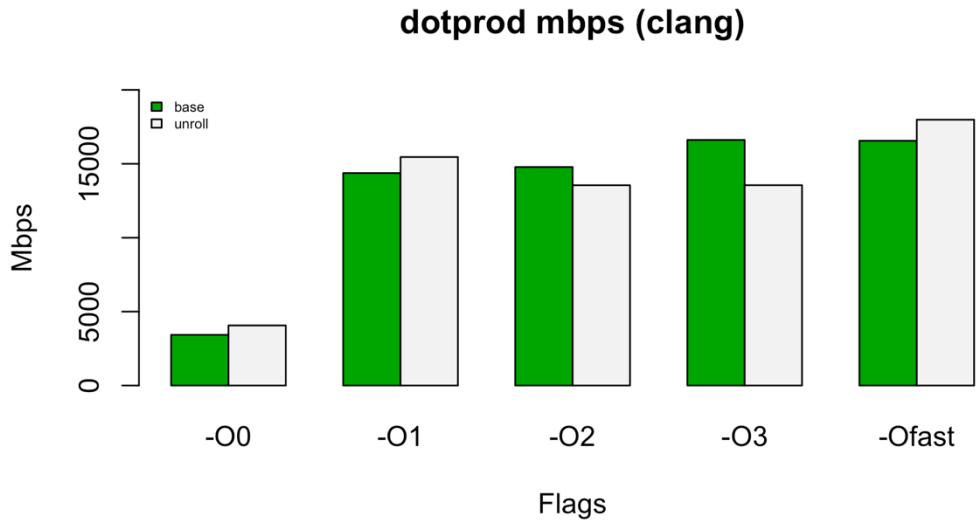


Pour BASE et UNROLL, généralement quand le flag d'optimisation s'élève, *mbps* du programme augmente, mais ce n'est pas vrai pour -O2.

Pour chaque flag, *mbps* de UNROLL est généralement plus élevée que celle de BASE, mais ce n'est pas vrai pour -O2.

Donc négligeant -O0, -O2 UNROLL a une *mbps* plus base et -Ofast UNROLL a une *mbps* plus élevée

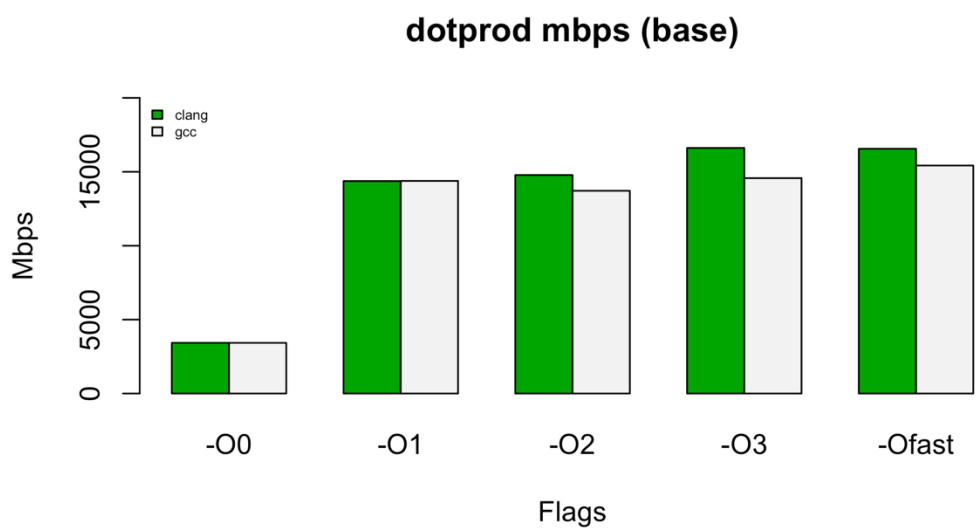
CLANG	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O0	BASE	0.125	0	0	8	1000	94.414	17.698	18.07	17.808	0.078(0.436%)	3427.488
O0	UNROLL	0.125	0	0	8	1000	178.689	14.983	15.106	15.016	0.023(0.154%)	4064.658
CLANG	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O1	BASE	0.125	0	0	8	1000	460.535	4.225	4.437	4.247	0.036(0.844%)	14370.64
O1	UNROLL	0.125	0	0	8	1000	125.639	3.937	3.979	3.948	0.009(0.228%)	15458.224
CLANG	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O2	BASE	0.125	0	0	8	1000	244.672	3.663	4.232	4.129	0.203(4.923%)	14781.416
O2	UNROLL	0.125	0	0	8	1000	35.49	4.494	4.558	4.505	0.011(0.252%)	13549.317
CLANG	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O3	BASE	0.125	0	0	8	1000	221.43	3.659	3.867	3.674	0.035(0.954%)	16611.219
O3	UNROLL	0.125	0	0	8	1000	190.225	4.491	4.535	4.503	0.009(0.193%)	13552.964
CLANG	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
Ofast	BASE	0.125	0	0	8	1000	94.86	3.662	3.954	3.686	0.062(1.682%)	16559.732
Ofast	UNROLL	0.125	0	0	8	1000	385.585	3.384	3.43	3.394	0.008(0.239%)	17984.055

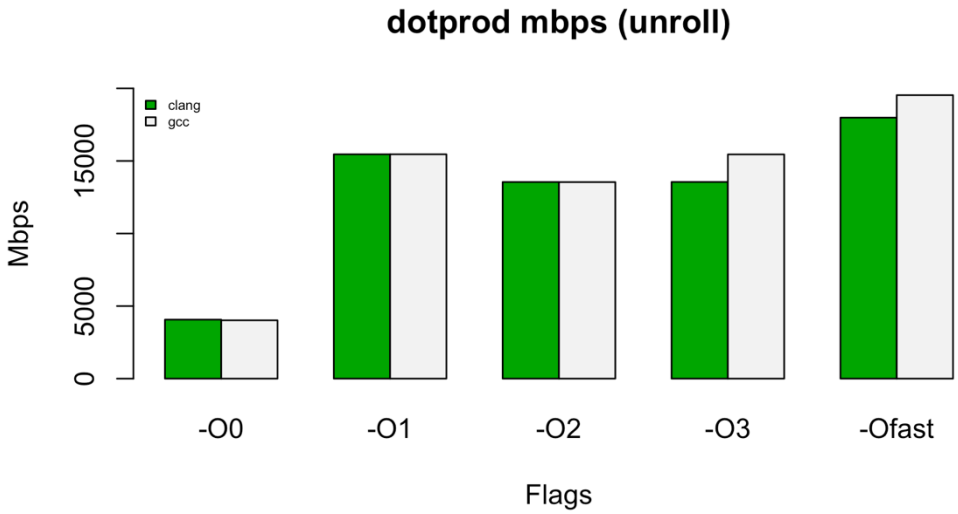


Pour BASE, quand le flag d'optimisation s'élève, la *mbps* du programme augmente en général. Pour UNROLL, *mbps* de -O1, -Ofast sont les plus enlevées, en revanche *mbps* de -O2, -O3 sont les plus bases.
Pour -O2, -O3 les *mbps* de BASE sont plus élevées que celles de UNROLL, sinon ans les autres cas c'est l'inverse.
Donc négligeant -O0, -O2 et -O3 UNROLL ont les plus bases *mbps*, -Ofast a la plus élevée *mbps*

Conclusion :
Dans les deux cas de compilateurs, les performances de UNROLL sont plus évidentes : -O2 UNROLL a la plus base *mbps*, à priori -O3 UNROLL l'a aussi, sinon -Ofast UNROLL a la plus élevée *mbps*. En revanche, les *mbps* de BASE ne sont pas homogènes quand nous changeons les compilateurs. Cela nous amène à s'intéresser plus précisément aux différences de *mbps* engendrées par des compilateurs différents.

- **Comparaison de mbps (Mib/s) entre compilateur CLANG et GCC**





D'abord nous observons que CLANG génère les versions de BASE dont *mbps* sont plus élevées avec les flags -O2,-O3, -Ofast. Puis nous observons que GCC génère les versions de UNROLL dont *mbps* sont plus élevées avec les flags -O3,-Ofast. Dans d'autres cas, les différences de *mbps* sont négligeables.

Conclusion :

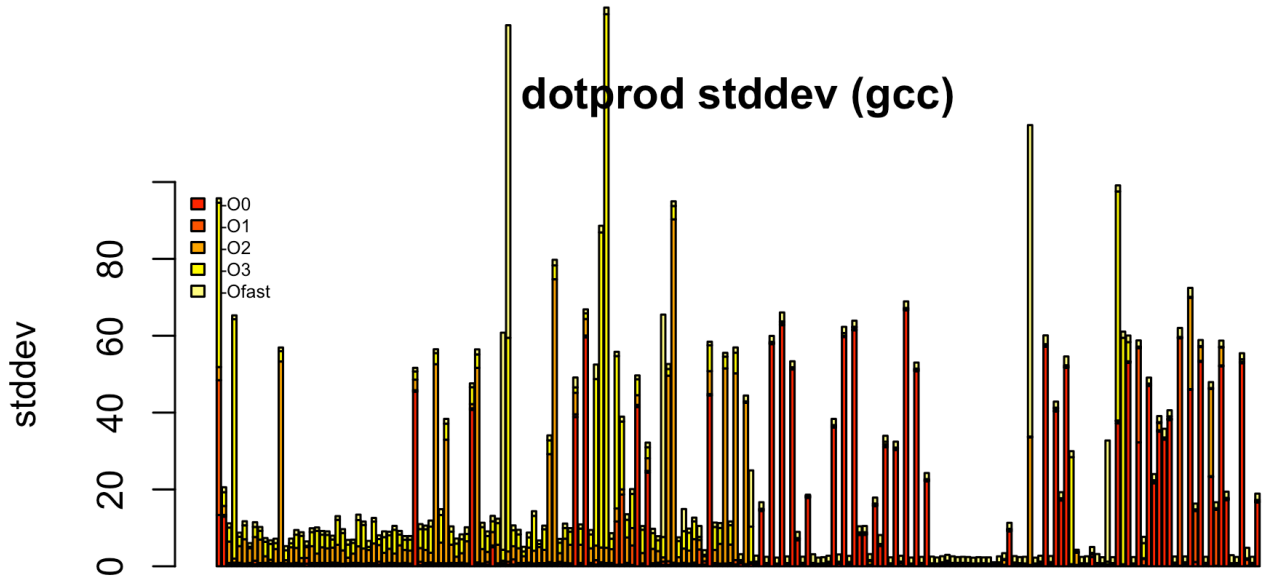
Alors, analysons pourquoi BASE est plus performant avec CLANG et UNROLL est plus performant avec GCC en consultant le site COMPILER EXPLORER.

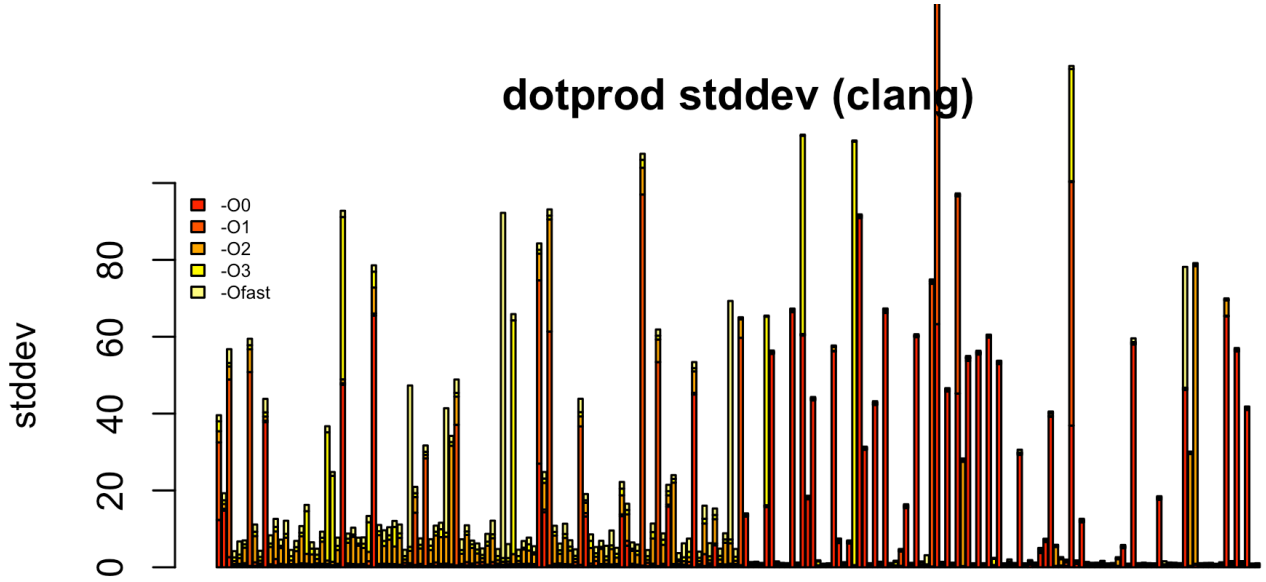
D'un côté, nous pouvons prétendre que cela liée au fait que les optimisations de CLANG comparées à GCC sont trop agressives avec les flags tels que -O2,-O3,-Ofast qui font des déroulages. Donc ce déroulage contribue à la *mbps* de BASE qui n'est pas déroulé, mais pour UNROLL qui est déjà déroulé, les inconvénients d'un extra-déroulage compensent les avantages d'un déroulage. Par exemple, que le pipeline soit rempli et il y a encore des instructions qui ne soient pas dans le pipeline, nous perdons ainsi le parallélisme. D'un autre côté, GCC ne fait pas de déroulages, mais une vectorisation avec les flags -O3 et -Ofast, il est donc plus cohérent avec UNROLL.

A noter : une vectorisation des *SD* permet d'accélérer le programme, car un registre *xmm* fait 128 bits, or un *SD* est 64 bits, donc en stockant un *SD* dans un *xmm*, nous prenons que la moitié de l'espace de *xmm*, l'autre est gaspillé. Donc en stockant deux *SD* dans un *xmm* et en faisant les opérations avec *PD*, nous bénéficions tous les espaces de *xmm*

- **Comparaison de stddev (%)**

La 1ere moitie de histogrammes sont les 101 résultats de BASE, l'autre moitié de histogrammes sont ceux de UNROLL





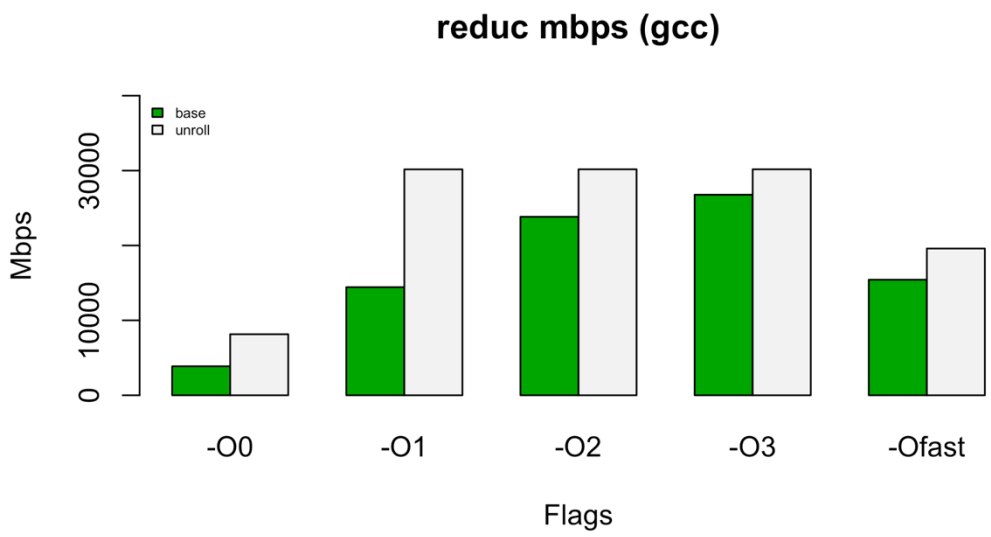
Visuellement les histogrammes en rouges sont moins nombreux et moins élevés dans le 1er graphe, nous prétendons qu'alors les versions compilées par GCC sont plus stables, notamment celles générées par les flags -O0 et -O1.
Donc à priori, dans les cas comme ci-dessus, où les *mbps* de CLANG -O0/-O1 et GCC -O0/-O1 sont presque identiques, nous choisirons GCC comme notre compilateur pour faire une compilation avec les flags -O0,-O1.

➢ Résultats de reduc

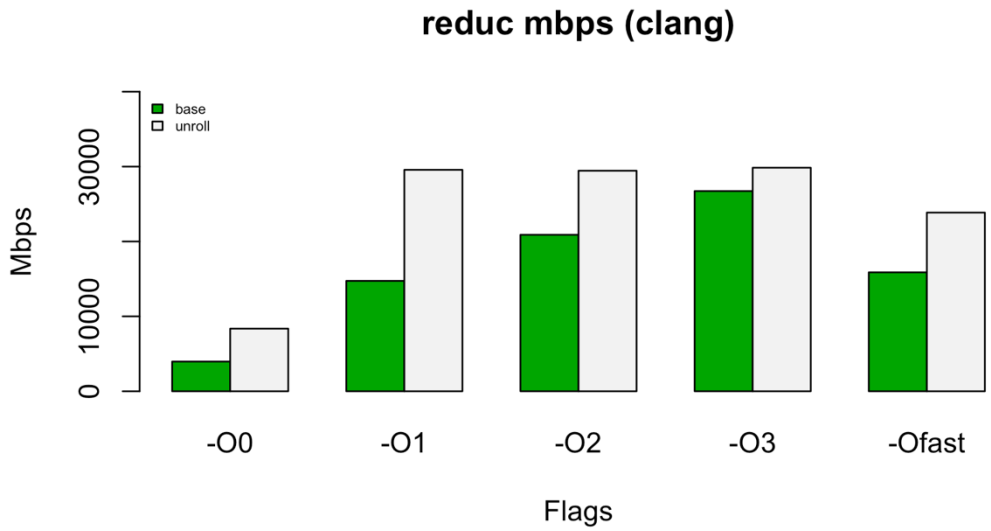
❖ Comparaison de mbps (Mib/s) entre version BASE et UNROLL

Vu que les résultats obtenus par GCC et CLANG sont très homogènes, nous les décrirons ensemble :

GCC	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O0	BASE	0.125	0	0	8	1000	36.933	15.693	16.092	15.761	0.066(0.421%)	3872.521
O0	UNROLL	0.125	0	0	8	1000	35.947	7.454	7.663	7.488	0.034(0.455%)	8151.227
GCC	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O1	BASE	0.125	0	0	8	1000	16.453	4.214	4.345	4.229	0.022(0.516%)	14432.527
O1	UNROLL	0.125	0	0	8	1000	559.876	2.013	2.079	2.024	0.011(0.548%)	30162.935
GCC	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O2	BASE	0.125	0	0	8	1000	37.386	2.547	2.639	2.561	0.015(0.581%)	23829.729
O2	UNROLL	0.125	0	0	8	1000	17.173	2.013	2.058	2.023	0.008(0.377%)	30171.52
GCC	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O3	BASE	0.125	0	0	8	1000	14.853	2.266	2.454	2.28	0.032(1.383%)	26773.72
O3	UNROLL	0.125	0	0	8	1000	28.274	2.013	2.063	2.023	0.009(0.444%)	30171.972
GCC	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
Ofast	BASE	0.125	0	0	8	1000	21.642	3.94	4.187	3.956	0.042(1.059%)	15428.857
Ofast	UNROLL	0.125	0	0	8	1000	20.125	3.107	3.137	3.115	0.006(0.187%)	19596.238



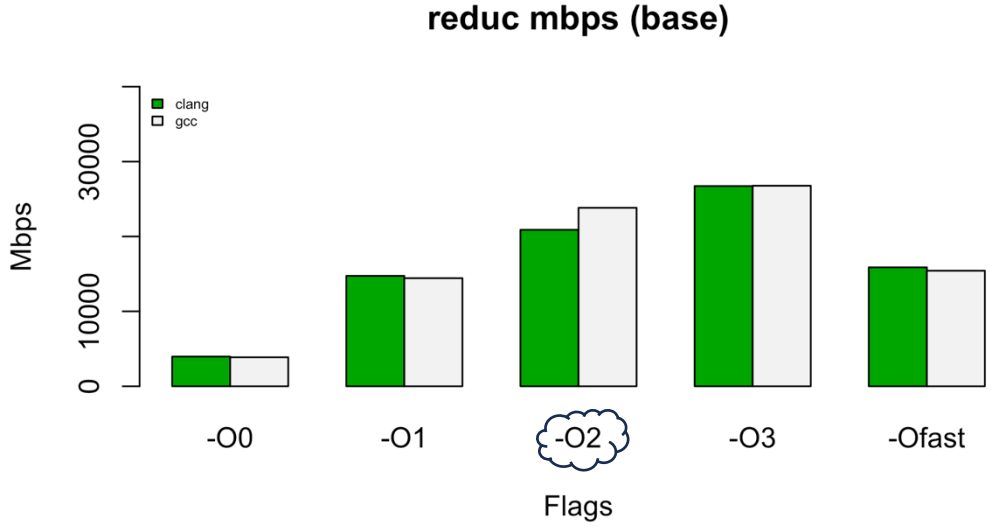
CLANG	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O0	BASE	0.125	0	0	8	1000	105.659	15.339	15.701	15.402	0.059(0.383%)	3962.877
O0	UNROLL	0.125	0	0	8	1000	22.488	7.272	7.446	7.3	0.030(0.406%)	8361.397
CLANG	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O1	BASE	0.125	0	0	8	1000	56.854	3.94	4.524	4.141	0.168(4.068%)	14738.692
O1	UNROLL	0.125	0	0	8	1000	13.635	2.03	2.279	2.064	0.045(2.193%)	29569.126
CLANG	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O2	BASE	0.125	0	0	8	1000	91.367	2.827	3.396	2.921	0.111(3.806%)	20898.113
O2	UNROLL	0.125	0	0	8	1000	68.265	2.03	2.494	2.073	0.088(4.267%)	29449.8
CLANG	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
O3	BASE	0.125	0	0	8	1000	21.859	2.268	2.464	2.283	0.033(1.437%)	26735.338
O3	UNROLL	0.125	0	0	8	1000	43.628	2.028	2.271	2.045	0.042(2.045%)	29847.811
CLANG	title	KiB	MiB	GiB	n	r	d	min	max	mean	stddev(%)	MiB/s
Ofast	BASE	0.125	0	0	8	1000	19.149	3.807	4.492	3.845	0.119(3.104%)	15874.653
Ofast	UNROLL	0.125	0	0	8	1000	288.112	2.55	2.584	2.558	0.006(0.250%)	23860.216

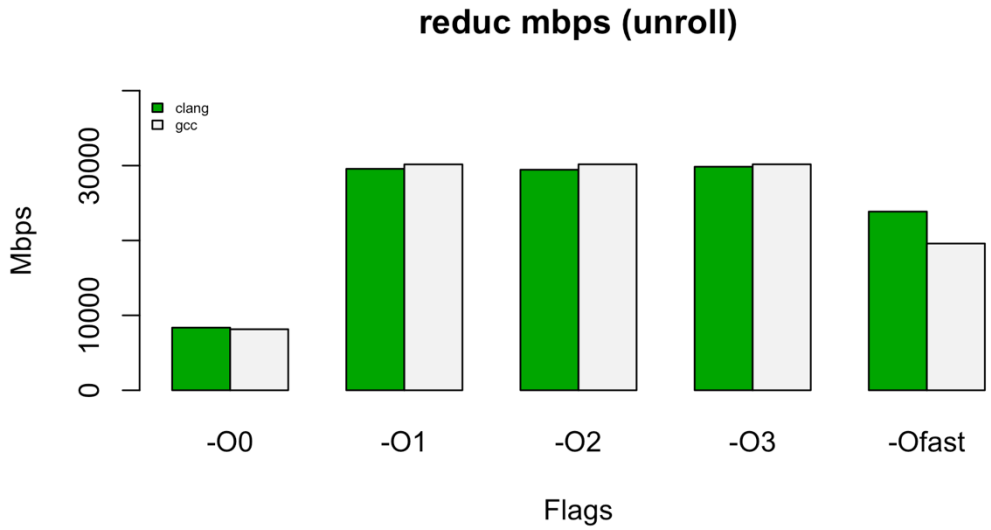


Pour BASE, quand le flag d'optimisation est plus s'élève, *mbps* du programme augmente monotonement jusqu'à -O3 et retombe avec -Ofast. Pour UNROLL, *mbps* du programme atteint son pic avec -O1, -O2, -O3, puis retombe aussi avec -Ofast. Pour tous les flags, les performances de UNROLL sont meilleures que celles de BASE
Donc négligeant -O0, -O1, -O2 et -O3 UNROLL ont les plus élevées *mbps*, -Ofast BASE a la plus base *mbps*.

Conclusion :
il parait que nous obtenons exactement les mêmes valeurs de *mbps* avec ces deux compilateurs, les *mbps* de UNROLL et BASE sont toutes très homogènes. Cela nous amène à s'intéresser plus précisément aux différences de *mbps* engendrées par les compilateurs différents.

- **Comparaison de mbps (Mib/s) entre compilateur CLANG et GCC**



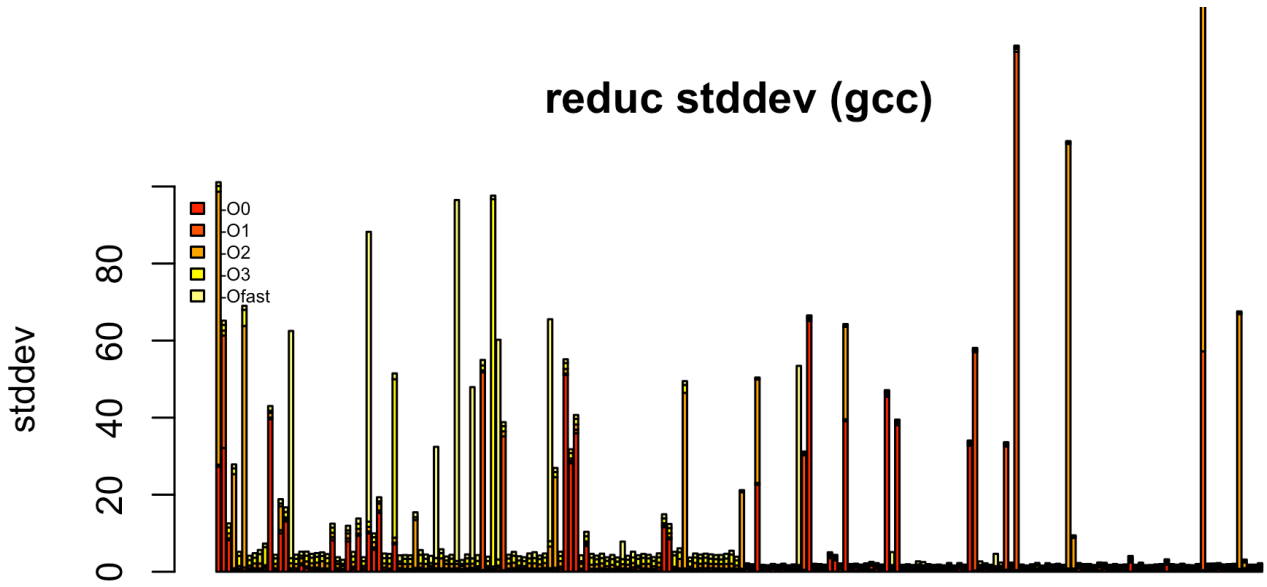


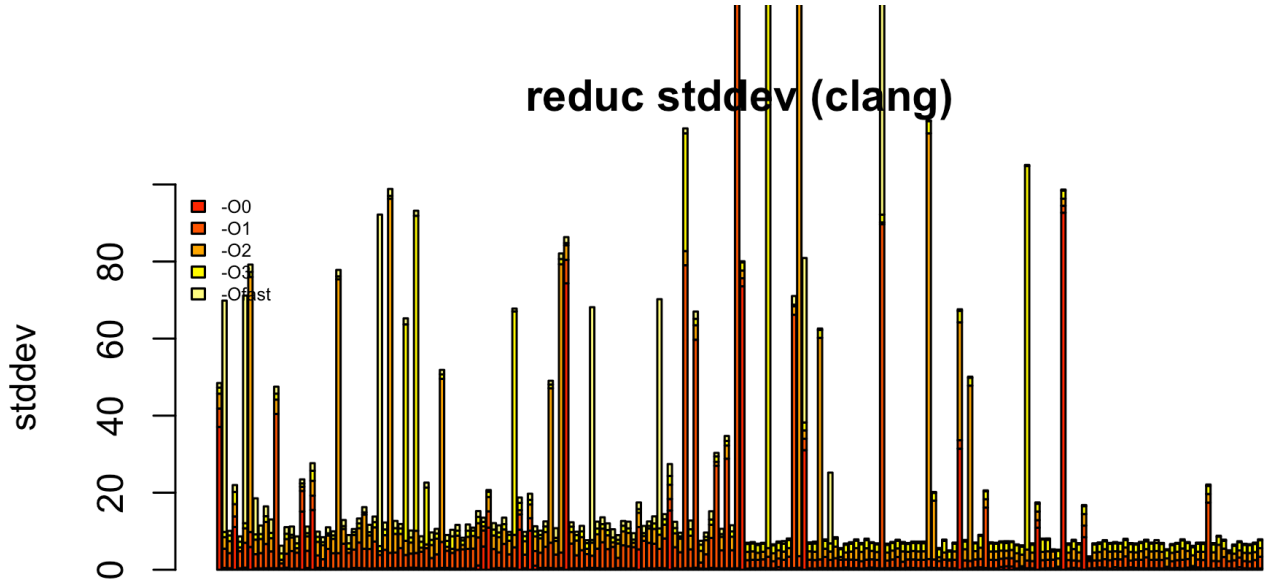
D'abord nous observons que CLANG génère les versions de BASE plus performantes avec la plupart de flags, sauf qu'avec le flag -O2, GCC -O2 BASE est plus performante que CLANG -O2 BASE.
Puis nous observons que GCC génère les versions de UNROLL plus performantes avec les flags -O1,-O2,-O3, sauf qu'avec le flag -Ofast, CLANG -Ofast UNROLL est plus performante que GCC -Ofast UNROLL.

Conclusion :
Comme *dotprod*, en regardant toutes les versions en langage assembleur, ces différences sont principalement engendrées par la vectorisation et le déroulage. Plus précisément par le fait que GCC fait le déroulage si nous le faisons à la main, sinon il ne le fait pas et que CLANG ne fait pas la vectorisation (sauf -Ofast) alors que GCC la fait.
Cependant, nous constatons deux comportements en particulier. D'abord, contrairement aux cas de *dotprod*, CLANG -Ofast garde un déroulage x8 comme ce que nous avons envie de faire, il ne fait pas un extra-déroulage en plus du déroulage x8 voulu. Il fait bien sûr aussi une vectorisation, mais avec le mem flag -Ofast, GCC ne fait que la vectorisation, donc évidement CLANG -Ofast UNROLL est plus rapide. Puis, les résultats marqués avec des nuages (P.8, P12) rassurent que le déroulage ne donne pas la meilleure performance, même si c'est un déroulage x8 voulu, il y un autre moyen qui n'est pas non plus la vectorisation, de produire un programme plus performant au niveau de vitesse.

- **Comparaison de stddev (%)**

La 1ere moitie de histogrammes sont les 101 résultats de BASE, l'autre moitié de histogrammes sont ceux de UNROLL





Visuellement les histogrammes sont moins nombreux et moins élevés dans le 1er graphe quel que soit la couleur, nous prétendons qu'alors les versions compilées par GCC sont plus stables en général.

IX. Conclusion

Pour faciliter l'enchainement de lecture et la compréhension, les conclusions sont déjà faites au fur et à mesure.
Dû à un temps limité, certains résultats comme le nombre d'histogrammes comptés pour la comparaison de *stddev*, les versions en langage assembleur ne sont pas tous fournis dans le rapport, mais nous trouverons surement les mêmes résultats en refaisant les démarches. Nous pouvons aller encore plus loin sur de nombreux résultats, par exemple d'étudier les cas marqués avec les nuages et de trouver les changements qui permettent une telle *mbps*.

X. Outils et Scripts

Pour compiler les programmes et les exécuter, j'ai écrit un scripte *exe.sh*. Pour trier les résultats obtenus, j'ai écrit un autre script *pars.sh*. En manquant le temps pour écrire un README, je vous fournis une simple présentation:

- Dans chaque répertoire *dotprod* et *reduc*, vous trouverez 2 scripts *exe.sh*, *pars.sh*, 1 répertoire *graphe* contenant *graphe.Rmd.*
- *exe.sh* : fait les compilations et exécute 101 fois le programme, sauvegarde les résultats dans 2 répertoires *gcc* et *clang*, sauvegarde les exécutables dans un sous répertoire *build*. Il ne fait RIEN sur les résultats obtenus. A faire particulièrement attention qu'en le réexécutant, vous perdrez TOUS les résultats précédents !
- *pars.sh* : trie les résultats obtenus, sauvegarde les résultats tries dans 2 sous répertoires *base_8* et *unroll_8* des répertoires *gcc* et *clang*, et dans répertoires *gcc* et *clang* aussi. Il sauvegarde tous les fichiers intermédiaires dans le sous répertoire *build* des répertoires *base_8* et *unroll_8*.
- *graphe.Rmd* : génère des tableaux en csv qui serviront à la construction des graphes. Vous pouvez exécuter les instructions par block souhaité, il n'y aura pas le problème sur les affectations de variables.
- Ne pas modifier la structure des répertoires ! Ne pas déplacer les fichiers ! Ne pas changer les noms de répertoires ni de fichiers ! Le path de script est trouvé automatiquement, ne pas inquiéter !
- Quelques préventions et messages d'indications sont implémentés dans les scripts pour éviter de mauvaises manipulations. Cliquer simplement sur *exe.sh* puis *pars.sh* (dans l'ordre), en fin ouvrir *graphe.Rmd* et RunAll.

J'ai fait des graphes avec GNUplot, mais la lecture des instructions est trop impraticable pour moi. J'ai en fin utilisé R que j'ai appris en L3, dont les instructions sont plus lisibles et permet de créer des données sous forme csv.

XI. Références et Liens utiles

"AP/TP2 at Main · Yaspr/AP." *GitHub*, github.com/yaspr/AP/tree/main/TP2.

"Barplot Function | R Documentation." *Www.rdocumentation.org*, www.rdocumentation.org/packages/graphics/versions/3.6.2/topics/barplot.

"CLang Optimizations.md." *Gist*, gist.github.com/lolo32/fd8ce29b218ac2d93a9e.

"Compiler Explorer." *Godbolt.org*, godbolt.org.

"Gnuplot to Group Multiple Bars." *Stack Overflow*, stackoverflow.com/questions/10783770/gnuplot-to-group-multiple-bars.

"Optimize Options (Using the GNU Compiler Collection (GCC))." *Gcc.gnu.org*, gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

"Pipeline (Architecture Des Processeurs)." *Wikipedia*, 30 Apr. 2022, fr.wikipedia.org/wiki/Pipeline_(architecture_des_processeurs).

"Reliable Way for a Bash Script to Get the Full Path to Itself." *Stack Overflow*, stackoverflow.com/questions/4774054/reliable-way-for-a-bash-script-to-get-the-full-path-to-itself.

"菜鸟教程 - 学的不仅是技术，更是梦想！." *Www.runoob.com*, www.runoob.com. For shell commands, for scripts.