

UNIVERSITÉ SAINT QUENTIN EN YVELINES
UNIVERSITÉ PARIS SACLAY



RAPPORT PPN

M1 CALCUL HAUTE HAUTE PERFORMANCE, SIMULATION

Utilisation et analyse critique des rapports MAQAO pour optimiser des mini-apps

Étudiants:

Anis MEHIDI
Arezki Takfarines HAMIDANI
Katia MOUALI
Madjid BOUZOURENE
Sylia BENBACHIR

Responsables:

Cédric Valensi
Emmanuel Oseret

13 Janvier 2022

Table des matières

1	Introduction	4
2	C'est quoi MAQAO ?	4
3	L'outil MAQAO ONEVIEW	4
4	Différents flags de compilation utilisé	5
5	Maqao sur un cas simple : Nbody3D	6
5.1	Code source	6
5.1.1	Makefile	6
5.1.2	Nbody.c	6
5.2	Déroulement du processeur d'optimisation	9
5.2.1	Étape Initiale	9
5.2.2	Version 1 corrigé de Nbody3D	14
5.2.3	Version 2 corrigé de Nbody3D	18
5.3	Comparaison des rapports	22
6	Critique de MAQAO	23
6.1	Les points fort de MAQAO	23
6.2	Les points négatives de MAQAO	25
7	Conclusion	27

Table des figures

1	Global Metrics de la version de base.	9
2	Experiment Summary de la version de base.	10
3	Loops Index de la version de base.	10
4	Rapport CQA de la version de base.	11
5	Gain : Code clean check de la version de base.	11
6	Gain : Vectorization de la version de base.	12
7	Potential : FMA de la version de base.	12
8	Hint : Unroll opportunity de la version de base.	13
9	Summary :Stylizer de la version de base.	13
10	Global Metrics de la version optimisé.	16
11	Loops Index de la version 1 optimisé.	17
12	Summary : Stylizer de la version 1 optimisé.	17
13	Gain de la version 1 optimisé.	17
14	Vector unaligned load/store instructions de la version 1 optimisé.	18
15	Global Metrics de la version 2 optimisé Finale.	20
16	Summary : Stylizer de la version 2 optimisé Finale.	21
17	Loops Index de la version 2 optimisé Finale.	21
18	Rapport CQA de la version 2 optimisé Finale.	21
19	Comparaison entre les rapports NBody3D.	22
20	Nbody2D 1ere version modification seulement des flags de compilation.	23
21	Nbody2D Information sur la machine.	23
22	Suggestion de MAQAO de permuter les boucles.	24
23	Comparaison entre avant et après permutations des boucles.	24
24	L'alignement mémoire fait.	25
25	La sortie CQA pour posix_memealign.	25
26	FMA NBody3D Avant.	26
27	FMA NBody3D Après	26
28	Erreur fautes de frappe	27

1 Introduction

Dans le monde de la programmation informatique, l'optimisation est le processus qui consiste à réduire le temps d'exécution d'une fonction, l'espace occupé par les données et le programme, ou la consommation d'énergie.

En règle générale, l'optimisation doit s'effectuer une fois que le programme est fonctionnel et qu'il réponde aux spécificités attendues.

Avant de commencer l'optimisation, pour cela, il existe plusieurs approches d'optimisation l'une plus complexe que l'autre, on peut citer quelques unes :

- Au niveau algorithmique, en choisissant un algorithme de complexité inférieure (au sens mathématique) et des structures de données adaptées,
- Au niveau du langage de développement, en ordonnant au mieux les instructions et en utilisant les bibliothèques disponibles,
- En utilisant localement un langage de bas niveau, qui peut être le langage C ou, pour les besoins les plus critiques, le langage assembleur.

Comme on peut le voir tout cela demande énormément de temps et de travail. C'est là qu'intervient **MAQAO** (Modular Assembly Quality Analyzer and Optimizer) qui est de façon générale un outil d'analyse et d'optimisation des performances.

Dans ce qui suit, nous allons définir plus en détails ce qu'est ce MAQAO , l'outil d'analyse Oneview, les différents flags de compilation ainsi qu'un code qu'on va optimiser grâce à MAQAO.

2 C'est quoi MAQAO ?

MAQAO (Modular Assembly Quality Analyzer and Optimizer) est un framework qui permet d'analyser et d'optimiser les performances d'un programme grâce à un ensemble de modules (CQA, LPROF, ONEVIEW). Cet outil effectue une analyse dynamique et statique sur le binaire du code source, pour déterminer les éléments limitant la performance d'une zone de l'application. L'objectif principal de MAQAO est de guider les développeurs d'applications tout au long du processus d'optimisation grâce à des rapports synthétiques et des astuces.

3 L'outil MAQAO ONEVIEW

ONEVIEW est un des modules délivré par MAQAO, permet de générer des rapport d'analyse de codes source compilés il se base sur les autre module de MAQAO (LPROF, CQA). Oneview propose différents formats pour pouvoir lire nos rapports comme HTML(par défaut), XSLX ou text. Son utilisation est relativement simple, elle consiste en une ligne de commande composée de différents champs :

maqao oneview $\underbrace{-create - report = < report >}_1$ $\underbrace{-c = < config >}_2$ $\underbrace{[-xp = < dir >]}_3$
 $\underbrace{[-of = < format >]}_4$ $\underbrace{[- - with - scalability]}_5$

1. **-create-report=<report>** permet de réaliser un rapport et exécutez toutes les étapes nécessaires les valeurs disponibles sont : one.
2. **-c=<config>** spécifie le chemin d'un fichier de configuration.
3. **-xp=<dir>** spécifie le chemin d'un répertoire d'expérimentation.

4. **-of=<format>** spécifie le format du rapport généré (HTML pas défaut) .
5. **-with-scalability** active l'analyse d'évolutivité.

Comme on peut aussi comparer des rapports entre eux grâce à Oneview en utilisant cette commande :

maqao oneview $\underbrace{- - compare - reports}_1 - - inputs = \underbrace{\underbrace{< xp1 >}_3, \underbrace{< xp2 >}_4}_2 \dots$

1. **-compare-reports** permet de réaliser une comparaison entre deux ou plusieurs rapports MAQAO.
2. **-inputs=** spécifie les noms des répertoires à ce qu'on doit faire la comparaison des rapports.
3. **<xp1>** spécifie le nom du répertoire 1.
4. **<xp2>** spécifie le nom du répertoire 2.

Il existent encore plusieurs options pour ONEVIEW, pour ce faire, on doit exécuter la commande suivante :

maqao oneview -help

4 Différents flags de compilation utilisé

-g	Produit les informations de débogage pour le débogueur GDB GNU.
-funroll-loops	Déroule les boucles dont le nombre d'itérations peut être déterminé à la compilation ou à l'entrée dans la boucle.
-O2	Niveau d'optimisation plus élevé. Temps de compilation plus lent, mieux pour les builds de production.
-O3	Niveau d'optimisation plus élevé. Son problème est que le temps de compilation est plus lent et la taille du binaire est importante.
-Ofast	Permet un niveau d'optimisation plus élevé que (-O3). Il active beaucoup de flags comme : -ffast-math ...
-fassociative-math	Autoriser la réassociation d'opérandes dans une série d'opérations à virgule flottante.
-ftree-vectorize	Effectuer une vectorisation sur les arbres.
-floop-unroll-and-jam	Appliquez des transformations de déroulement et de bourrage sur des boucles réalisables. Activé par défaut dans -O3

5 Maqao sur un cas simple : Nbody3D

Afin de réaliser notre rapport PPN, il était nécessaire de comprendre le fonctionnement des différents flags d'optimisation des compilateurs de MAQAO et d'apprendre à l'utiliser. Pour cela, nous avons cherché à optimiser un benchmark déjà vu en cours : **Nbody3D**. L'idée était de faire le travail d'optimisation un maximum de fois depuis la version de base afin d'identifier les informations récurrentes et exploitable qui nous étaient produites.

5.1 Code source

5.1.1 Makefile

La version de base du Makefile qui nous était donnée, il est compilé et exécuté sans aucun flag d'optimisation.

```
1 all: nbody.g
2 nbody.g: nbody.c
3     gcc -g -mavx2 -fopt-info-all=nbody.gcc.optprt $< -o $@ -lm -fopenmp
4 clean:
5     rm -Rf *~ nbody.g *.optprt
```

5.1.2 Nbody.c

```
1 #include <omp.h>
2 #include <math.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 //
7 typedef float          f32;
8 typedef double         f64;
9 typedef unsigned long long u64;
10 //
11 typedef struct particle_s {
12     f32 x, y, z;
13     f32 vx, vy, vz;
14 } particle_t;
15 //
16 void init(particle_t *p, u64 n)
17 {
18     for (u64 i = 0; i < n; i++)
19     {
20         //
21         u64 r1 = (u64)rand();
22         u64 r2 = (u64)rand();
23         f32 sign = (r1 > r2) ? 1 : -1;
24         //
25         p[i].x = sign * (f32)rand() / (f32)RAND_MAX;
26         p[i].y = (f32)rand() / (f32)RAND_MAX;
27         p[i].z = sign * (f32)rand() / (f32)RAND_MAX;
28         //
29         p[i].vx = (f32)rand() / (f32)RAND_MAX;
30         p[i].vy = sign * (f32)rand() / (f32)RAND_MAX;
31         p[i].vz = (f32)rand() / (f32)RAND_MAX;
```

```

32     }
33 }
34 //
35 void move_particles(particle_t *p, const f32 dt, u64 n)
36 {
37     //
38     const f32 softening = 1e-20;
39     //
40     for (u64 i = 0; i < n; i++)
41     {
42         //
43         f32 fx = 0.0;
44         f32 fy = 0.0;
45         f32 fz = 0.0;
46
47         //23 floating-point operations
48         for (u64 j = 0; j < n; j++)
49         {
50             //Newton's law
51             const f32 dx = p[j].x - p[i].x; //1
52             const f32 dy = p[j].y - p[i].y; //2
53             const f32 dz = p[j].z - p[i].z; //3
54             const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening; //
55             9
56             const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0); //11
57             //Net force
58             fx += dx / d_3_over_2; //13
59             fy += dy / d_3_over_2; //15
60             fz += dz / d_3_over_2; //17
61         }
62         //
63         p[i].vx += dt * fx; //19
64         p[i].vy += dt * fy; //21
65         p[i].vz += dt * fz; //23
66     }
67     //3 floating-point operations
68     for (u64 i = 0; i < n; i++)
69     {
70         p[i].x += dt * p[i].vx;
71         p[i].y += dt * p[i].vy;
72         p[i].z += dt * p[i].vz;
73     }
74 }
75 //
76 int main(int argc, char **argv)
77 {
78     //
79     const u64 n = (argc > 1) ? atoll(argv[1]) : 16384;
80     const u64 steps= 10;
81     const f32 dt = 0.01;
82     //
83     f64 rate = 0.0, drate = 0.0;
84
85     //Steps to skip for warm up
86     const u64 warmup = 3;

```

```

86 //
87 particle_t *p = malloc(sizeof(particle_t) * n);
88 //
89 init(p, n);
90
91 const u64 s = sizeof(particle_t) * n;
92
93 printf("\n\033[1mTotal memory size:\033[0m %llu B, %llu KiB, %llu MiB
94 \n\n", s, s >> 10, s >> 20);
95 //
96 printf("\033[1m%5s %10s %10s %8s\033[0m\n", "Step", "Time, s", "
97 Interact/s", "GFLOP/s"); fflush(stdout);
98 //
99 for (u64 i = 0; i < steps; i++)
100 {
101     //Measure
102     const f64 start = omp_get_wtime();
103
104     move_particles(p, dt, n);
105     const f64 end = omp_get_wtime();
106     //Number of interactions/iterations
107     const f32 h1 = (f32)(n) * (f32)(n - 1);
108
109     //GFLOPS
110     const f32 h2 = (23.0 * h1 + 3.0 * (f32)n) * 1e-9;
111
112     if (i >= warmup)
113     {
114         rate += h2 / (end - start);
115         drate += (h2 * h2) / ((end - start) * (end - start));
116     }
117     //
118     printf("%5llu %10.3e %10.3e %8.1f %s\n",
119         i,
120         (end - start),
121         h1 / (end - start),
122         h2 / (end - start),
123         (i < warmup) ? "*" : "");
124     fflush(stdout);
125 }
126 //
127 rate /= (f64)(steps - warmup);
128 drate = sqrt(drate / (f64)(steps - warmup) - (rate * rate));
129
130 printf("-----\n");
131 printf("\033[1m%s %4s \033[42m%10.11f +- %.11f GFLOP/s\033[0m\n",
132     "Average performance:", "", rate, drate);
133 printf("-----\n");
134 //
135 free(p);
136 //
137 return 0;
138 }

```


5.2 Déroulement du processeur d'optimisation

5.2.1 Étape Initiale

Dans la première étape, nous allons effectuer une analyse binaire en utilisant les flags permettant à MAQAO de faire son analyse. Après ça, on va utiliser les informations obtenues afin d'améliorer les performances de notre programme en suivant les différentes indications du rapport MAQAO.

La page Index

On commence notre analyse par la page d'accueil du rapport. Comme on peut l'observer, on retrouve des informations qui sont utiles afin d'optimiser notre programme.

1. Global Metrics

On suit cette lecture afin de savoir ce qu'il faut faire :

- Pour les informations en vert : quand la valeur est bonne.
- Pour les informations en orange claire : quand la valeur est bonne mais qu'elle peut s'améliorer.
- Pour les informations en orange : lorsque la valeur est moyenne et montre un problème de performance potentiel.
- Pour les informations en rouge : lorsque la valeur est mauvaise et montre probablement un problème de performances, on doit impérativement les rajouter afin d'avoir un rapport efficace.

Global Metrics		
Total Time (s)		66.76
Profiled Time (s)		66.76
Time in analyzed loops (%)		37.9
Time in analyzed innermost loops (%)		37.9
Time in user code (%)		38.7
Compilation Options		nbody.g: -O2, -O3 or -Ofast is missing. -march=(target) is missing. -funroll-loops is missing.
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		83.3
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.06
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.12
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.55
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.21
	Nb Loops to get 80%	1

FIGURE 1 – Global Metrics de la version de base.

En analysant le Global Metrics de notre programme, on constate ces points :

- Que ce programme est compilé sans flags d'optimisation et on doit impérativement les utiliser pour avoir un programme optimisé.
- Que notre Array Access Efficiency est efficace qu'à 75%, alors qu'on devrait avoir un 100%.
- Que les Speedup peuvent-être meilleure si le programme est vectorisé à la compilation et ils doivent être à 1.
-

A cette étape, nous allons prendre en compte la suggestion des flags [O2, O3, Ofast], -march=target et -funroll-loops pour le prochain programme à produire.

2. Experiment Summary

Experiment Summary			
Application	.nbody.g		
Timestamp	2021-12-27 19:06:52	Universal Timestamp	1640628412
Number of processes observed	1	Number of threads observed	1
Experiment Type	Sequential		
Machine	anism-VivoBook-ASUSLaptop-X515EA-X515EA		
Model Name	11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz		
Architecture	x86_64	Micro Architecture	TIGER_LAKE
Cache Size	8192 KB	Number of Cores	4
OS Version	Linux 5.4.0-91-generic #102-Ubuntu SMP Fri Nov 5 16:31:28 UTC 2021		
Architecture used during static analysis	x86_64	Micro Architecture used during static analysis	TIGER_LAKE
Compilation Options	nbody.g: GNU 10.3.0 -mavx2 -mtune=generic -march=x86-64 -g -fopenmp -fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-protection -fcf-protection		

FIGURE 2 – Experiment Summary de la version de base.

Dans les options de compilation, on voit que le flag -march existe avec x86-64, car il utilise la version de base de -march et pour que notre programme soit optimisé, on doit le modifier à -march=target

La page Loops

A cette étape, nous savons que notre code est peut être optimisable et que les Speedups sont intéressants et que nous pouvons les améliorer, c'est dans cette étape qu'on va découvrir les modifications du code à effectuer.

Loops Index													
Filters													
Columns Filter													
<input checked="" type="checkbox"/> Level	<input checked="" type="checkbox"/> Coverage run_0 (%)	<input checked="" type="checkbox"/> Max Time Over Threads run_0 (s)	<input checked="" type="checkbox"/> Time w.r.t. Wall Time run_0 (s)	<input checked="" type="checkbox"/> Nb Threads run_0	<input checked="" type="checkbox"/> Vectorization Ratio (%)	<input checked="" type="checkbox"/> Vectorization Efficiency (%)	<input checked="" type="checkbox"/> Speedup If No Scalar Integer	<input checked="" type="checkbox"/> Speedup If FP Vectorized	<input checked="" type="checkbox"/> Speedup If Fully Vectorized	<input checked="" type="checkbox"/> Speedup If Perfect Load Balancing run_0	<input type="checkbox"/> Stride 0	<input type="checkbox"/> Stride 1	<input type="checkbox"/> Stride n
<input type="checkbox"/> Stride Unknown	<input type="checkbox"/> Stride Indirect	<input type="button" value="Select none"/>	<input type="button" value="Select All Coverages"/>										
Loop id	Source Location	Source Function	Level	Coverage run_0 (%)	Max Time Over Threads run_0 (s)	Time w.r.t. Wall Time run_0 (s)	Nb Threads run_0	Vectorization Ratio (%)	Vectorization Efficiency (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized	Speedup If Perfect Load Balancing run_0
1	nbody.g - nbody: c57-69	move_particles	Innermost	37.9	25.3	25.3	1	0	8.19	1.19	1.4	14.55	1

FIGURE 3 – Loops Index de la version de base.

Comme on peut le voir, on a un tableau récapitulatif des boucles que nous devons optimiser, ici dans notre cas on a une seule boucle qui nous pose problème, avec un Coverage de 37.9% et une vectorisation de 0% et des speedup que nous devons améliorer et qui doivent être à 1.

Pour savoir ce qu'il faut modifier exactement, MAQAO nous donne des étapes à suivre dans le rapport CQA.

Rapport CQA

Le rapport CQA se présente comme le montre la figure suivante : On peut voir le code

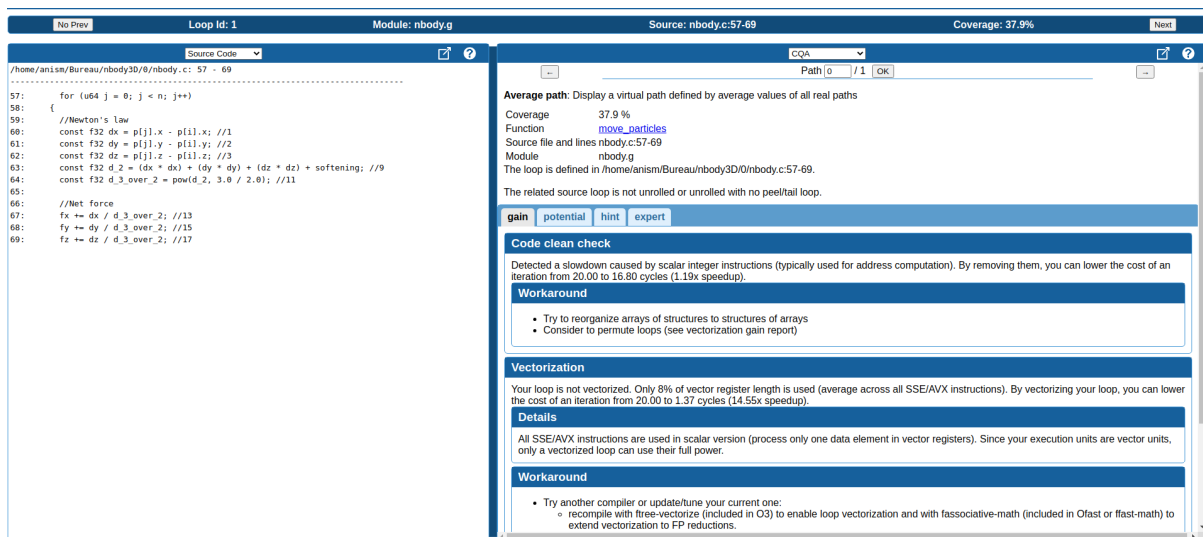


FIGURE 4 – Rapport CQA de la version de base.

source de la boucle sur la gauche et les améliorations à effectuée dessus sur la droite. Nous allons réaliser chacune des modifications demandée lorsque cela est possible afin de pouvoir constater dans une analyse ultérieure si notre programme est optimisé.

• Gain : Code clean check

Dans cette section, ça nous montre qu'il y'a un ralentissement causé par des instructions d'entier scalaire (généralement utilisées pour le calcul d'adresse). En les supprimant, nous pouvons réduire le coût d'une itération.

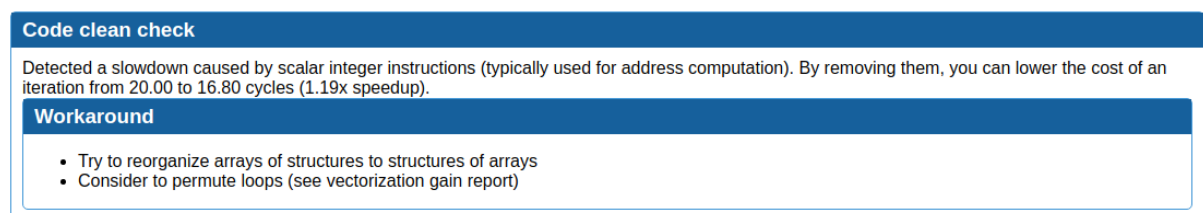


FIGURE 5 – Gain : Code clean check de la version de base.

- **Gain : Vectorization**

Dans cette section, on nous montre que notre boucle n'est pas du tout vectorisée, il nous montre les instructions à suivre afin d'avoir une vectorisation meilleure, en prenant en compte des flags d'optimisation **ftee-vectorize** et **fassociative-math** et aussi changer la structure du code qui est en AOS (Array of Structures) en SOA (Structures of Array).

Vectorization

Your loop is not vectorized. Only 8% of vector register length is used (average across all SSE/AVX instructions). By vectorizing your loop, you can lower the cost of an iteration from 20.00 to 1.37 cycles (14.55x speedup).

Details

All SSE/AVX instructions are used in scalar version (process only one data element in vector registers). Since your execution units are vector units, only a vectorized loop can use their full power.

Workaround

- Try another compiler or update/tune your current one:
 - recompile with `ftee-vectorize` (included in O3) to enable loop vectorization and with `fassociative-math` (included in Ofast or ffast-math) to extend vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make it unit-stride:
 - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly: C storage order is row-major: `for(i) for(j) a[j][i] = b[j][i];` (slow, non stride 1) => `for(i) for(j) a[i][j] = b[i][j];` (fast, stride 1)
 - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA): `for(i) a[i].x = b[i].x;` (slow, non stride 1) => `for(i) a.x[i] = b.x[i];` (fast, stride 1)

FIGURE 6 – Gain : Vectorization de la version de base.

- **Potential : FMA**

Dans cette section, on nous montre qu'on doit essayer de changer l'ordre dans lequel les éléments sont évalués (à l'aide de parenthèses) dans les expressions arithmétiques contenant à la fois les opérations ADD/SUB et MUL pour permettre à votre compilateur de générer des instructions FMA valide dans la mesure du possible et qu'on doit recompiler avec le flag **-march=tigerlake**

gain potential hint expert

FMA

Presence of both ADD/SUB and MUL operations.

Workaround

- Recompile with `march=tigerlake`. CQA target is `Tiger_lake_8c` (11th generation Intel Core processors based on Tiger Lake microarchitecture) but specialization flags are `-march=x86-64`
- Try to change order in which elements are evaluated (using parentheses) in arithmetic expressions containing both ADD/SUB and MUL operations to enable your compiler to generate FMA instructions wherever possible. For instance `a + b*c` is a valid FMA (MUL then ADD). However `(a+b)*c` cannot be translated into an FMA (ADD then MUL).

FIGURE 7 – Potential : FMA de la version de base.

- **Hint : Unroll opportunity**

Dans cette section, on nous montre qu'on doit rajouter des options **-funroll-loops** et/ou **-floop-unroll-and-jam** afin d'avoir un meilleur déroulage de boucle.

Unroll opportunity

Loop is data access bound.

Workaround

Unroll your loop if trip count is significantly higher than target unroll factor and if some data references are common to consecutive iterations. This can be done manually. Or by recompiling with `-funroll-loops` and/or `-floop-unroll-and-jam`.

FIGURE 8 – Hint : Unroll opportunity de la version de base.

La page Summary

Dans cette section, on nous montre qu'on doit rajouter des options d'optimisation et ce qu'on doit changer pour avoir un programme plus performant.

▼ Stylizer

Good

[4 / 4] Application is long enough (66.76 s)

To have good quality measurements, it is advised that the application execution time is greater than 10 seconds.

[2 / 2] Application is correctly profiled ("Others" category represents 0 % of the execution time)

To have a representative profiling, it is advised that the category "Others" represents less than 20% of the execution time in order to analyze as much as possible of the user code

Bad

[0 / 3] Optimization level option not used

To have better performances, it is advised to help the compiler by using a proper optimization level (-O2 or higher). Warning, depending on compilers, faster optimization levels can decrease numeric accuracy.

[0 / 3] Architecture specific option is not used

-march=x86-64 option is used but it is not specific enough to produce efficient code.

[2 / 3] Helper debug compilation option -fno-omit-frame-pointer is missing

-fno-omit-frame-pointer improves the accuracy of callchains found during the application profiling.

► Strategizer

FIGURE 9 – Summary :Stylizer de la version de base.

5.2.2 Version 1 corrigé de Nbody3D

Makefile corrigé

```
1  all: nbody.g nbody.gl
2
3  nbody.g: nbody.c
4      gcc -g -mavx2 -fopt-info-all=nbody.gcc.opttrpt $< -o $@ -lm -fopenmp
5
6  nbody.gl: nbody1.c
7      gcc -mavx2 -funroll-loops -march=tigerlake -mtune=generic -finline-
      functions -fno-omit-frame-pointer -ftree-vectorize -Ofast -g -fopt-info-
      all=nbody.gcc.opttrpt $< -o $@ -lm -fopenmp
8
9  clean:
10      rm -Rf *~ nbody.g nbody.gl *.opttrpt
```

Code source corrigé

Arrivé a cette étape, les principaux changement de notre code source sont les suivants : changement de AOS en SOA et enroulement d'une boucle comme le montre le code source suivant :

Changement en l'ordre de SOA de la fonction init

```
1
2  void init(particle_t p, u64 n)
3  {
4
5      for (u64 i = 0; i < n; i++)
6      {
7          //
8          u64 r1 = (u64)rand();
9          u64 r2 = (u64)rand();
10         f32 sign = (r1 > r2) ? 1 : -1;
11
12         //
13         p.x[i] = sign * (f32)rand() * (1/(f32)RAND_MAX);
14         p.y[i] = (f32)rand() * (1/(f32)RAND_MAX);
15         p.z[i] = sign * (f32)rand() * (1/(f32)RAND_MAX);
16
17         //
18         p.vx[i] = (f32)rand() * (1/(f32)RAND_MAX);
19         p.vy[i] = sign * (f32)rand() * (1/(f32)RAND_MAX);
20         p.vz[i] = (f32)rand() * (1/(f32)RAND_MAX);
21
22     }
23 }
```

Changement en l'ordre de SOA de la fonction move_particles

```
1 void move_particles(particle_t p, const f32 dt, u64 n)
2 {
3     //
4     const f32 softening = 1e-20;
5
6     //
7     for (u64 i = 0; i < n; i++)
8     {
9         //
10        f32 fx = 0.0;
11        f32 fy = 0.0;
12        f32 fz = 0.0;
13
14        //23 floating-point operations
15        for (u64 j = 0; j < n; j++)
16        {
17            //Newton's law
18            const f32 dx = p.x[j] - p.x[i]; //1
19            const f32 dy = p.y[j] - p.y[i]; //2
20            const f32 dz = p.z[j] - p.z[i]; //3
21            const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening; //9
22            const f32 d_3_over_2 = pow(d_2, 1.5); //11
23
24
25            //Net force
26            fx += (dx * (1/d_3_over_2)); //13
27            fy += (dy * (1/d_3_over_2)); //15
28            fz += (dz * (1/d_3_over_2)); //17
29        }
30
31
32        //
33        p.vx[i] += (dt * fx); //19
34        p.vy[i] += (dt * fy); //21
35        p.vz[i] += (dt * fz); //23
36    }
37
38    //3 floating-point operations
39    for (u64 i = 0; i < n; i++)
40    {
41        p.x[i] += (dt * p.vx[i]);
42        p.y[i] += (dt * p.vy[i]);
43        p.z[i] += (dt * p.vz[i]);
44    }
45 }
46
```

Allocation de la mémoire

```

1  particle_t p;
2  p.x = malloc(n * sizeof(f32));
3  p.y = malloc(n * sizeof(f32));
4  p.z = malloc(n * sizeof(f32));
5  p.vx = malloc(n * sizeof(f32));
6  p.vy = malloc(n * sizeof(f32));
7  p.vz = malloc(n * sizeof(f32));

```

Libérer les allocations

```

1  free(p.x);
2  free(p.y);
3  free(p.z);
4  free(p.vx);
5  free(p.vy);
6  free(p.vz);

```

Résultats obtenues

Après exécution du programme, on remarque notre programme est amélioré avec des speedup de 1.0, une vectorisation à 100%, les options de compilation qui sont complètes, comme on le voit dans les figures suivantes :

1. **La page index** : On voit clairement que notre programme est totalement vectorisé et plus efficace que la version de base : gain de temps d'exécution constaté par rapport à toutes la version de base précédente et d'autres meilleures paramètres.

Global Metrics ?	
Total Time (s)	2.05
Profiled Time (s)	2.05
Time in analyzed loops (%)	100
Time in analyzed innermost loops (%)	100
Time in user code (%)	100
Compilation Options	OK
Perfect Flow Complexity	1.00
Array Access Efficiency (%)	99.9
Perfect OpenMP + MPI + Pthread	1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.00
No Scalar Integer	Potential Speedup
	Nb Loops to get 80%
	1
FP Vectorised	Potential Speedup
	Nb Loops to get 80%
	1
Fully Vectorised	Potential Speedup
	Nb Loops to get 80%
	1
FP Arithmetic Only	Potential Speedup
	Nb Loops to get 80%
	1

FIGURE 10 – Global Metrics de la version optimisé.

2. **La page Loops** On voit que notre programme est vectorisé mais qu'a 93% en Vectorization Efficiency. De ce fait, on consulte son rapport CQA afin de voir les modifications qu'il faut apporter.

Loop id	Source Location	Source Function	Level	Coverage run_0 (%)	Max Time Over Threads run_0 (s)	Time w.r.t. Wall Time run_0 (s)	Nb Threads run_0	Vectorization Ratio (%)	Vectorization Efficiency (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized	Speedup If Perfect Load Balancing run_0	Stride 0	Stride 1	Stride n	Stride Unknown	Stride Indirect
4	nbody.g1 - nbo move_partic dy1.c:60-73 les		Innermost	99.76	2.05	2.05	1	100	93.1	1	1	1	1	0	3	0	0	0
2	nbody.g1 - nbo move_partic dy1.c:84-88 les		Single	0.24	0	0	1	0	6.25	1	1	16	0	0	1	34	1	0

FIGURE 11 – Loops Index de la version 1 optimisé.

3. La page Summary

Dans cette section, après avoir rajouter tout les flags d'optimisation demandé, on a eu une nouvelle erreur, et pour corriger ça , faudra mettre le n qu'on exécute de façon qu'on doit dépasser les 10 seconde d'exécution.

Stylizer	Stratizer
<p>Good</p> <p>[3 / 3] Optimization level option -Ofast is used To have better performances, it is advised to help the compiler by using a proper optimization level (-Ofast)</p> <p>[3 / 3] Architecture specific option -march=tigerlake is used</p> <p>[3 / 3] Helper debug compilation options -g and -fno-omit-frame-pointer are used -g option gives access to debugging informations, such as source locations and -fno-omit-frame-pointer improve the accuracy of callchains found during the application profiling.</p> <p>[2 / 2] Application is correctly profiled ("Others" category represents 0 % of the execution time) To have a representative profiling, it is advised that the category "Others" represents less than 20% of the execution time in order to analyze as much as possible of the user code</p>	<p>Bad</p> <p>[0 / 4] Application is too short (2.05 s) If the overall application execution time is less than 10 seconds, many of the measurements at function or loop level will very likely be under the measurement quality threshold (0.1 seconds). Rerun to increase runtime duration: for example use a larger dataset or include a repetition loop.</p>

FIGURE 12 – Summary : Stylizer de la version 1 optimisé.

3. Rapport CQA

- **Gain :** On voit que notre boucle est vectorisé, mais que 93% de la longueur du registre.

gain	potential	hint	expert
<p>Vectorization</p> <p>Your loop is vectorized, but using 93% register length (average across all SSE/AVX instructions).</p> <p>Details</p> <p>All SSE/AVX instructions are used in vector version (process two or more data elements in vector registers).</p>			
<p>Execution units bottlenecks</p> <p>Performance is limited by execution of divide and square root operations (the divide/square root unit is a bottleneck). By removing all these bottlenecks, you can lower the cost of an iteration from 96.00 to 52.00 cycles (1.85x speedup).</p> <p>Workaround</p> <ul style="list-style-type: none"> Reduce the number of division or square root instructions: <ul style="list-style-type: none"> If denominator is constant over iterations, use reciprocal (replace x/y with $x*(1/y)$). Check precision impact. This will be done by your compiler with <code>ffast-math</code> or <code>Ofast</code> If you accept to loose numerical precision up to 2 ulp, you can speedup your code by passing the following options to your compiler: (<code>ffast-math</code> or <code>Ofast</code>) and <code>mrecip</code> 			

FIGURE 13 – Gain de la version 1 optimisé.

- **Hint : Vector unaligned load/store instructions**

D'après les sections du rapport suivante, on se rend compte que notre programme est vectorisé mais pas totalement. Ce problème vient du fait que notre mémoire n'est pas alignée et nous appliquons donc la suggestion d'utiliser la fonction **posix_memalign**.

Vector unaligned load/store instructions
Detected 4 suboptimal vector unaligned load/store instructions.
Details <ul style="list-style-type: none"> • VEXTRACTF128: 2 occurrences • VINSERTF128: 2 occurrences
Workaround <p>Use vector aligned instructions:</p> <ol style="list-style-type: none"> 1. align your arrays on 64 bytes boundaries: replace { void *p = malloc (size); } with { void *p; posix_memalign (&p, 64, size); }. 2. inform your compiler that your arrays are vector aligned: if array 'foo' is 64 bytes-aligned, define a pointer 'p_foo' as <code>__builtin_assume_aligned (foo, 64)</code> and use it instead of 'foo' in the loop.

FIGURE 14 – Vector unaligned load/store instructions de la version 1 optimisé.

5.2.3 Version 2 corrigé de Nbody3D

Code source corrigé

Changement de l'enroulement de la boucle avec un pas de 4 dans la fonction init

```

1  for (u64 i = 0; i < n; i+=4)
2  {
3      //
4      u64 r1 = (u64)rand();
5      u64 r2 = (u64)rand();
6      f32 sign = (r1 > r2) ? 1 : -1;
7
8      //
9      p.x[i] = sign * (f32)rand() * (1/(f32)RAND_MAX);
10     p.y[i] = (f32)rand() * (1/(f32)RAND_MAX);
11     p.z[i] = sign * (f32)rand() * (1/(f32)RAND_MAX);
12
13     //
14     p.vx[i] = (f32)rand() * (1/(f32)RAND_MAX);
15     p.vy[i] = sign * (f32)rand() * (1/(f32)RAND_MAX);
16     p.vz[i] = (f32)rand() * (1/(f32)RAND_MAX);
17
18     //
19     p.x[i+1] = sign * (f32)rand() * (1/(f32)RAND_MAX);
20     p.y[i+1] = (f32)rand() * (1/(f32)RAND_MAX);
21     p.z[i+1] = sign * (f32)rand() * (1/(f32)RAND_MAX);
22
23     //
24     p.vx[i+1] = (f32)rand() * (1/(f32)RAND_MAX);
25     p.vy[i+1] = sign * (f32)rand() * (1/(f32)RAND_MAX);
26     p.vz[i+1] = (f32)rand() * (1/(f32)RAND_MAX);
27
28     //
29     p.x[i+2] = sign * (f32)rand() * (1/(f32)RAND_MAX);
30     p.y[i+2] = (f32)rand() * (1/(f32)RAND_MAX);
31     p.z[i+2] = sign * (f32)rand() * (1/(f32)RAND_MAX);

```

```

32
33 //
34 p.vx[i+2] = (f32)rand() * (1/(f32)RAND_MAX);
35 p.vy[i+2] = sign * (f32)rand() * (1/(f32)RAND_MAX);
36 p.vz[i+2] = (f32)rand() * (1/(f32)RAND_MAX);
37
38
39 //
40 p.x[i+3] = sign * (f32)rand() * (1/(f32)RAND_MAX);
41 p.y[i+3] = (f32)rand() * (1/(f32)RAND_MAX);
42 p.z[i+3] = sign * (f32)rand() * (1/(f32)RAND_MAX);
43
44 //
45 p.vx[i+3] = (f32)rand() * (1/(f32)RAND_MAX);
46 p.vy[i+3] = sign * (f32)rand() * (1/(f32)RAND_MAX);
47 p.vz[i+3] = (f32)rand() * (1/(f32)RAND_MAX);
48 }

```

Changement de l'enroulement de la boucle avec un pas de 4 dans la fonction move_particle

```

1 //3 floating-point operations
2 for (u64 i = 0; i < n; i+=4)
3 {
4     p.x[i] += (dt * p.vx[i]);
5     p.y[i] += (dt * p.vy[i]);
6     p.z[i] += (dt * p.vz[i]);
7
8     p.x[i+1] += (dt * p.vx[i+1]);
9     p.y[i+1] += (dt * p.vy[i+1]);
10    p.z[i+1] += (dt * p.vz[i+1]);
11
12    p.x[i+2] += (dt * p.vx[i+2]);
13    p.y[i+2] += (dt * p.vy[i+2]);
14    p.z[i+2] += (dt * p.vz[i+2]);
15
16    p.x[i+3] += (dt * p.vx[i+3]);
17    p.y[i+3] += (dt * p.vy[i+3]);
18    p.z[i+3] += (dt * p.vz[i+3]);
19
20 }

```

Changement de la définition de POW dans la fonction move_particle

```

1 //On change de POW vers SQRT car POW consomme le double de SQRT
2 const f32 t=sqrt(d_2);
3 const f32 d_3_over_2 = t*t*t; //11

```

Changement de la définition de notre allocation mémoire

```

1 //On utilise le posix_memalign
2 double * p_x = NULL;
3 double * p_y = NULL;
4 double * p_z = NULL;
5 double * p_vx = NULL;
6 double * p_vy = NULL;
7 double * p_vz = NULL;
8 int l = 0;
9 l += posix_memalign ((void **) &p_x, 64, n* sizeof(particle_t));
10 l += posix_memalign ((void **) &p_y, 64, n* sizeof(particle_t));
11 l += posix_memalign ((void **) &p_z, 64, n* sizeof(particle_t));
12 l += posix_memalign ((void **) &p_vx, 64, n* sizeof(particle_t));
13 l += posix_memalign ((void **) &p_vy, 64, n* sizeof(particle_t));
14 l += posix_memalign ((void **) &p_vz, 64, n* sizeof(particle_t));
15 if ( l ) return 1;
16 p_x = __builtin_assume_aligned(p_x, 64);
17 p_y = __builtin_assume_aligned(p_y, 64);
18 p_z = __builtin_assume_aligned(p_z, 64);
19 p_vx = __builtin_assume_aligned(p_vx, 64);
20 p_vy = __builtin_assume_aligned(p_vy, 64);
21 p_vz = __builtin_assume_aligned(p_vz, 64);

```

Résultats obtenues

Arrivé à la dernière étape de notre rapport, on voit clairement que notre programme est totalement vectorisé et est au plus efficace possible par rapport à toutes les versions précédentes et utilisation de tous les registres SSE/AVX.

1. La page index :

On voit clairement que notre programme est amélioré au niveau du Profiled Time.

Global Metrics		?
Total Time (s)		11.20
Profiled Time (s)		11.20
Time in analyzed loops (%)		100.0
Time in analyzed innermost loops (%)		99.9
Time in user code (%)		100.0
Compilation Options		OK
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		100
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.00
	Nb Loops to get 80%	1

FIGURE 15 – Global Metrics de la version 2 optimisé Finale.

2. La page Summary

Dans cette section, après avoir modifier la valeur de n avec laquelle on doit exécuter notre programme, on voit qu'on a plus d'erreur dans cette section.



FIGURE 16 – Summary : Stylizer de la version 2 optimisé Finale.

3. La page Loops

Loops Index

Filters

Columns Filter

☒ Level

☒ Coverage run_0 (%)

☒ Max Time Over Threads run_0 (s)

☒ Time w.r.t. Wall Time run_0 (s)

☒ Nb Threads run_0

☒ Vectorization Ratio (%)

☒ Vectorization Efficiency (%)

☒ Speedup If No Scalar Integer

☒ Speedup If FP Vectorized

☒ Speedup If Fully Vectorized

☒ Speedup If Perfect Load Balancing run_0

☐ Stride 0

☐ Stride 1

☐ Stride n

☐ Stride Unknown

☐ Stride Indirect

Select none

Select All Coverages

Select All Times

Loop id	Source Location	Source Function	Level	Coverage run_0 (%)	Max Time Over Threads run_0 (s)	Time w.r.t. Wall Time run_0 (s)	Nb Threads run_0	Vectorization Ratio (%)	Vectorization Efficiency (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized	Speedup If Perfect Load Balancing run_0
4	nbody.g2 - nbody2.c:89-103	move_particle	innermost	99.87	11.18	11.18	1	100	100	1	1	1	1

FIGURE 17 – Loops Index de la version 2 optimisé Finale.

4. Rapport CQA

Vectorization	
Your loop is fully vectorized, using full register length.	
Details	
All SSE/AVX instructions are used in vector version (process two or more data elements in vector registers).	

FIGURE 18 – Rapport CQA de la version 2 optimisé Finale.

5.3 Comparaison des rapports

Afin de vous illustrer la différence qu'il y'a entre les différents flags d'optimisation **-O2**, **-O3** et **-Ofast**, on a opter pour l'utilisation de l'outil de comparaison de rapports MAQAO comme la montre cette figure ci dessus :

► Compared Reports					
Global Metrics					
Metric	r1	r2	r3	r4	
Total Time (s)	304.29	68.65	72.38	12.20	
Profiled Time (s)	304.29	68.65	72.38	12.20	
Time in analyzed loops (%)	90.7	100.0	100	100.0	
Time in analyzed innermost loops (%)	90.7	100.0	100.0	99.9	
Time in user code (%)	93.3	100.0	100	100.0	
Compilation Options	nbody.g: -O2, -O3 or -Ofast is missing.	OK	OK	OK	
Perfect Flow Complexity	1.00	12.5	1.00	1.00	
Array Access Efficiency (%)	83.3	71.2	100	100	
Perfect OpenMP + MPI + Pthread	1.00	1.00	1.00	1.00	
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution	1.00	1.00	1.00	1.00	
No Scalar Integer	Potential Speedup 1.19	1.08	1.11	1.00	
	Nb Loops to get 80% 1	1	1	1	
FP Vectorised	Potential Speedup 1.20	1.41	1.47	1.00	
	Nb Loops to get 80% 1	1	1	1	
Fully Vectorised	Potential Speedup 6.27	12.6	11.9	1.00	
	Nb Loops to get 80% 1	1	1	1	
Only FP Arithmetic	Potential Speedup 1.69	1.21	1.31	1.00	
	Nb Loops to get 80% 1	1	1	1	

FIGURE 19 – Comparaison entre les rapports NBody3D.

6 Critique de MAQAO

6.1 Les points fort de MAQAO

Les flags de compilation

MAQAO nous a beaucoup aidé lors du choix des options de compilations. Voici un exemple où on a juste changer les flags de compilations, en gardent le même code et la même machine.

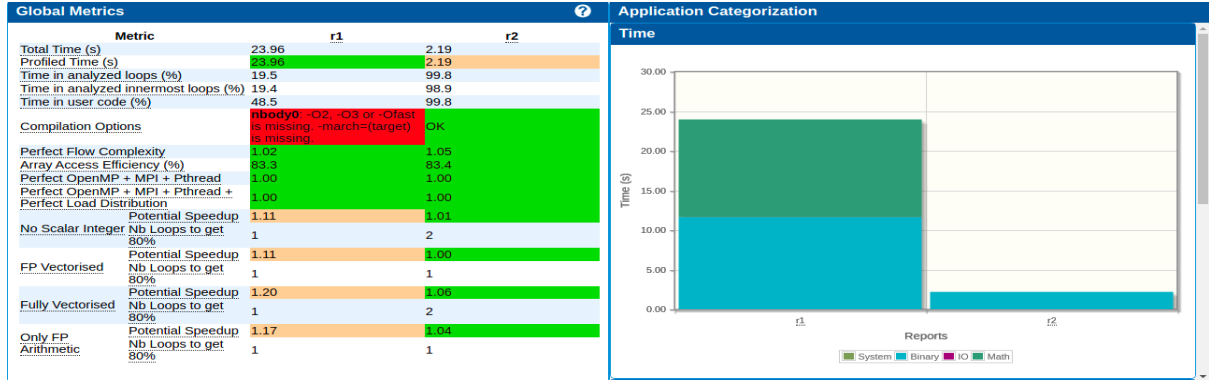


FIGURE 20 – Nbody2D 1ere version modification seulement des flags de compilation.

On a ganger x10 la vitesse d'exécutions par rapport à sans les bons flags de compilations.

Experiment Summaries			
	r1	r2	
Application	/nbody0	same as r1	
Timestamp	2021-12-28 12:48:01	2022-01-07 02:17:06	
Experiment Type	Sequential	same as r1	
Machine	madjid-Inspiron-3543	same as r1	
Architecture	x86_64	same as r1	
Micro Architecture	BROADWELL	same as r1	
Model Name	Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz	same as r1	
Cache Size	3072 KB	same as r1	
Number of Cores	2	same as r1	
Maximal Frequency	2.7 GHz	same as r1	
OS Version	Linux 5.11.0-43-generic #47~20.04.2-Ubuntu SMP Mon Dec 13 11:06:56 UTC 2021	same as r1	
Architecture used during static analysis	x86_64	same as r1	
Micro Architecture used during static analysis	BROADWELL	same as r1	
Compilation Options	nbbody0: GNU 9.3.0 -mtune=generic -march=x86-64 -g -funroll-loops -finline-functions -ftree-vectorize -fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-protection -fcf-protection	nbbody0: GNU 9.3.0 --param l1-cache-size=32 --param l1-cache-line-size=64 --param l2-cache-size=3072 -mtune=broadwell -mavx2 -march=broadwell -g -Ofast -funroll-loops -finline-functions -ftree-vectorize -fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-protection -fcf-protection	
Number of processes observed	1	same as r1	
Number of threads observed	1	same as r1	
MAQAO version	2.15.0	same as r1	
MAQAO build	b1544c69c095a29fedd570ae9a5f2917b3fb35a8::20211209-173719	same as r1	

FIGURE 21 – Nbody2D Information sur la machine.

Changement de l'ordre des boucles

Voici une boucle de multiplication de matrices

```

1 void mul_matrix ( int ** matrix_a , int ** matrix_b , int ** matrix_ab , int l
2 )
3 {
4   for ( int i = 0; i < l ; i ++ ) {
5     for ( int j = 0; j < l ; j ++ ) {
6       matrix_ab [i][j] = 0;
7       for ( int k = 0; k < l ; k ++ ) {
8         matrix_ab [i][j] += matrix_a [i][k] * matrix_b [k][j];
9       }
10    }
11  }
12 }

```

Workaround

- Try another compiler or update/tune your current one
- Remove inter-iterations dependences from your loop and make it unit-stride:
 - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly: C storage order is row-major: `for(i) for(j) a[j][i] = b[j][i];` (slow, non stride 1) => `for(i) for(j) a[i][j] = b[i][j];` (fast, stride 1)

FIGURE 22 – Suggestion de MAQAO de permuter les boucles.

```

1 void mul_matrix ( int ** matrix_a , int ** matrix_b , int ** matrix_ab , int l
2 )
3 {
4   for ( int i = 0; i < l ; i ++ ) {
5     for ( int k = 0; k < l ; k ++ ) {
6       for ( int j = 0; j < l ; j ++ ) {
7         matrix_ab [ i ][ j ] = 0;
8         matrix_ab [ i ][ j ] += matrix_a [ i ][ k ] * matrix_b [ k ][ j ];
9       }
10    }
11  }
12 }

```

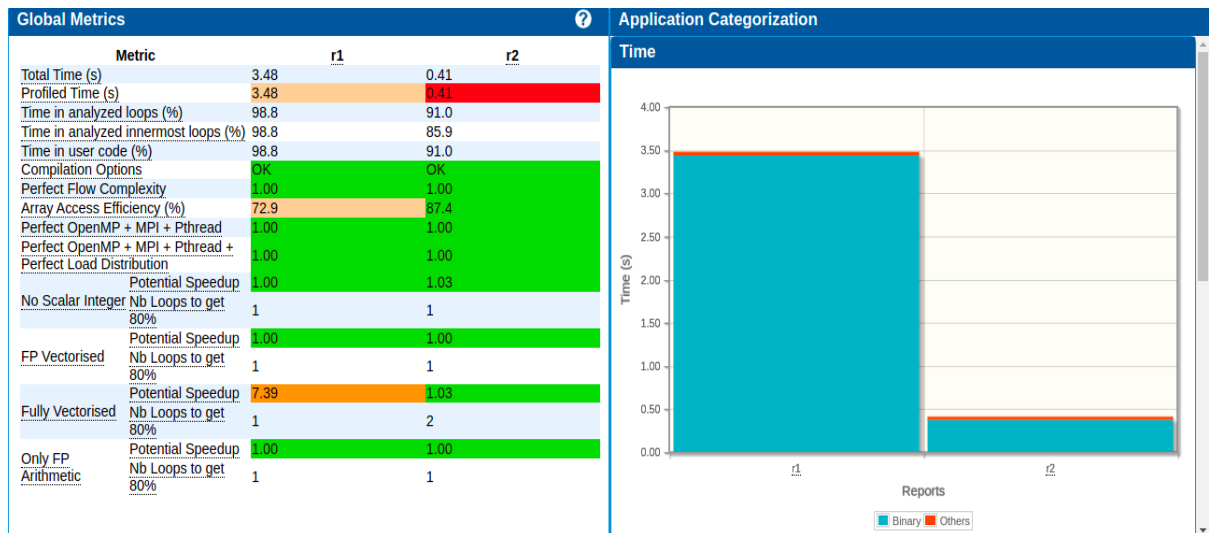


FIGURE 23 – Comparaison entre avant et après permutations des boucles.

Remarque : On remarque que lorsque on change seulement l'ordre des boucles, on a gagné en temps d'exécution et en vectorisation.

Les autres points fort

MAQAO nous a permis d'optimiser le programme de Nbody3D qu'on a illustré au début de notre rapport avec toute les indications dans les différents onglet, changements de structure de AoS à SoA, l'ajout des bons flags de compilation et les indications sur la vectorisation.

6.2 Les points négatives de MAQAO

Malgré ces points fort, MAQAO nous a pas aidé dans tous ses suggestions même des fois ils nous a induit en erreur ou ils nous demande de faire des modifications qu'on a déjà faite. Voici quelques exemples remarqué au cours de notre manipulation de MAQAO.

MAQAO et CQA

MAQAO ne se base pas sur le code source mais travaille exclusivement au niveau binaire, il est obligé un peu de deviner à quoi ressemble le code source et à quoi ressemble la compilation du coup ils savent pas si on a changé le code source tous ce qu'il regarde c'est le code assembleur qui résulte.

On peut avoir des cas ou il continue de nous demander d'essayer cette solution ou bien modifier cette boucle alors qu'en base la modification à était déjà faite.

Exemple : Si on change une partie dans le code source et que celle ci ne change pas son code assembleur, CQA n'affiche rien n'a changé. **Une limite de MAQAO.**

1. **Cas de `posix_memalign` :** Maqao nous a demandé d'aligner notre mémoire car elle n'était pas alignée en nous proposant de l'aligne avec la fonctions `posix_memealign`, malgré qu'on a fait le nécessaire pour ça, on a toujours cette suggestion au niveau de la dernière analyse.

```
double * p_x = NULL;
double * p_y = NULL;
double * p_z = NULL;
double * p_vx = NULL;
double * p_vy = NULL;
double * p_vz = NULL;
int l = 0;
l += posix_memalign ((void **) &p_x, 32, n* sizeof(particle_t));
l += posix_memalign ((void **) &p_y, 32, n* sizeof(particle_t));
l += posix_memalign ((void **) &p_z, 32, n* sizeof(particle_t));
l += posix_memalign ((void **) &p_vx, 32, n* sizeof(particle_t));
l += posix_memalign ((void **) &p_vy, 32, n* sizeof(particle_t));
l += posix_memalign ((void **) &p_vz, 32, n* sizeof(particle_t));
if ( l ) return 1;
p.x = __builtin_assume_aligned(p_x, 32);
p.y = __builtin_assume_aligned(p_y, 32);
p.z = __builtin_assume_aligned(p_z, 32);
p.vx = __builtin_assume_aligned(p_vx, 32);
p.vy = __builtin_assume_aligned(p_vy, 32);
p.vz = __builtin_assume_aligned(p_vz, 32);
```

FIGURE 24 – L'alignement mémoire fait.

Workaround

Use vector aligned instructions:

1. align your arrays on 32 bytes boundaries: replace { void *p = malloc (size); } with { void *p; posix_memalign (&p, 32, size); }.
2. inform your compiler that your arrays are vector aligned: if array 'foo' is 32 bytes-aligned, define a pointer 'p_foo' as `__builtin_assume_aligned (foo, 32)` and use it instead of 'foo' in the loop.

FIGURE 25 – La sortie CQA pour `posix_memealign`.

2. **Cas FMA (fused multiply-add) :** Malgré le changement de toute les multiplications et additions avec le format demandé, on retrouve toujours la même suggestion dans notre rapport.

```
for (u64 j = 0; j < n; j++)

//Newton's law
const f32 dx = p.x[j] - p.x[i]; //1
const f32 dy = p.y[j] - p.y[i]; //2
const f32 dz = p.z[j] - p.z[i]; //3
const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening; //
const f32 tmp=sqrt(d_2);
const f32 d_3_over_2 = d_2 * tmp; //11

//Net force
fx += (dx * (1/d_3_over_2)); //13
fy += (dy * (1/d_3_over_2)); //15
fz += (dz * (1/d_3_over_2)); //17
```

Average path: Display a virtual path defined by average values of all real paths

Coverage 100 %
Function [move_particles](#)
Source file and lines NbodyOpt.c:59-73
Module nbodyOp
The loop is defined in /home/madjid/Bureau/CHPS/Archi_Parallele/nbody3D/0/NbodyOpt.c:59-73.
The related source loop is not unrolled or unrolled with no peel/tail loop.

gain potential hint expert

FMA

Detected 80 FMA (fused multiply-add) operations. Presence of both ADD/SUB and MUL operations.

Workaround

Try to change order in which elements are evaluated (using parentheses) in arithmetic expressions containing both ADD/SUB and MUL operations to enable your compiler to generate FMA instructions wherever possible. For instance $a + b * c$ is a valid FMA (MUL then ADD). However $(a+b) * c$ cannot be translated into an FMA (ADD then MUL).

FIGURE 26 – FMA NBody3D Avant.

```
for (u64 j = 0; j < n; j++)
{
//Newton's law
const f32 dx = p.x[j] - p.x[i]; //1
const f32 dy = p.y[j] - p.y[i]; //2
const f32 dz = p.z[j] - p.z[i]; //3
const f32 d_2 = softening + dz * dz ;
const f32 d_3 = d_2 + dy * dy ;//9
const f32 d_4 = d_3 + dx * dx;
const f32 tmp=sqrt(d_4);
const f32 d_3_over_2 = d_4 * tmp; //11

//Net force
fx += (dx * (1/d_3_over_2)); //13
fy += (dy * (1/d_3_over_2)); //15
fz += (dz * (1/d_3_over_2)); //17
```

Average path: Display a virtual path defined by average values of all real paths

Coverage 99.85 %
Function [move_particles](#)
Source file and lines NbodyOpt.c:59-75
Module nbodyOp
The loop is defined in /home/madjid/Bureau/CHPS/Archi_Parallele/nbody3D/0/NbodyOpt.c:59-75.
The related source loop is not unrolled or unrolled with no peel/tail loop.

gain potential hint expert

FMA

Detected 80 FMA (fused multiply-add) operations. Presence of both ADD/SUB and MUL operations.

Workaround

Try to change order in which elements are evaluated (using parentheses) in arithmetic expressions containing both ADD/SUB and MUL operations to enable your compiler to generate FMA instructions wherever possible. For instance $a + b * c$ is a valid FMA (MUL then ADD). However $(a+b) * c$ cannot be translated into an FMA (ADD then MUL).

FIGURE 27 – FMA NBody3D Après .

3. **Aspect algorithmique :** MAQAO regarde l'aspect vectorisation et les opérations arithmétiques combien y'a d'addition, multiplication et division, il ne peut pas savoir en terme d'algorithmique combien au moins on peut avoir de calcul ADD, MUL, SUB et DIV.

Exemple :

```
const f32 tmp=sqrt(d_4);
const f32 d_3_over_2 = d_4 * tmp;
```

```
const f32 tmp=sqrt(d_4);
const f32 d_3_over_2 = tmp * tmp * tmp;
```

Dans cette exemple on aurait pas besoin de multiplier tmp trois fois il suffisé seulement de le multiplie une fois avec d_4 MAQAO n'arrive pas a simplifie les opération arithmétique.

4. Malgré que toute ces modifications que maqao nous a proposé, on a toujours pas eu un résultat de 100% au niveau de la Vectorization Efficiency, après recherche de notre coté on a trouvé que la puissance (POW) est une instruction complexe donc on du la changer à une racine carée (SQRT) pour régler notre problème de Vectorization Efficiency et avoir

un fully vectorized. Maqao nous a pas aidé dans cette étape pour pouvoir utiliser toute la longueur des registres, c'est par nous même que nous avons trouvé cette instruction complexe qu'on devait changer pour avoir un résultat plus performant.

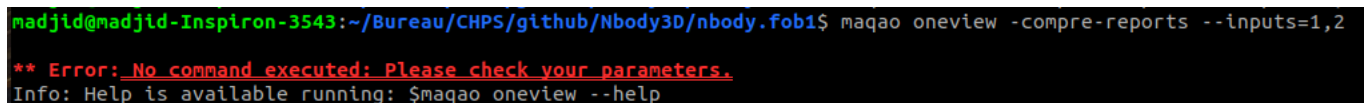
Incompréhension de l'origine de cette erreur :

```
1 PANIC : unprotected error in call to Lua API ([ string " lua_oneview "
2 ]:40457: attempt to index local __path__ ( a nil value ) )
```

Le programme s'exécute le plus normalement possible avec un temps d'exécution élevé mais cette erreur se manifeste et malheureusement ce n'est pas vraiment claire.

Remarque par rapport aux fautes de frappes dans l'invité de commandes

Dans la figure ci dessus on a une erreur comme quoi la commande n'existe pas et effectivement c'est le cas, mais on aurait aimé qu'au lieu de nous envoyer chercher la bonne commande dans **-help** c'est de suggérer une modification au niveau du terminale puisque il manque juste le "a" dans compare comme dans le langage de programmation C.



```
madjid@madjid-Inspiron-3543:~/Bureau/CHPS/github/Nbody3D/nbody.fob1$ maqao oneview -compre-reports --inputs=1,2
** Error: No command executed: Please check your parameters.
Info: Help is available running: $maqao oneview --help
```

FIGURE 28 – Erreur fautes de frappe

7 Conclusion

Ce projet avait comme objectif de manipuler et de critiquer afin d'avoir des améliorations pour MAQAO.

Nous avons testé plusieurs programmes pour cerner l'outil MAQAO et avoir une idée générale sur son utilisation et ces avantages et inconvénients.

Cet outil nous a permis d'améliorer les programmes tester en passant de tests très coûteux en temps et avec une vectorisation casé nulle à des programmes performants avec une différence de temps d'exécution considérable et une vectorisation complète. Mais malheureusement y'a des cas où MAQAO n'était pas très efficace comme mentionné dans la section critique.

Nous sommes assez certains, que cet outil, sera bénéfique et apportera du dynamisme et va permettre d'améliorer et d'optimiser du temps au utilisateurs afin d'analyser leurs programmes plus facilement.

Bibliographie

<http://www.maqao.org/>
http://www.maqao.org/release/MAQAO_QuickReferenceSheet_V12.pdf
<http://www.maqao.org/release/MAQAO.Tutorial.ONEVIEW.pdf>
<http://www.maqao.org/release/MAQAO.Tutorial.LProf.pdf>
<http://www.maqao.org/release/MAQAO.Tutorial.CQA.intel64.pdf>
<https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>