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
Madjid bouzourene
Arezki Takfarines Hamidani
Katia Mouali
Sylia Benbachir

Responsables:
Emmanuel Oseret
Cédric Valensi

11 Janvier 2022

Table des matières

Table des figures

1 Introduction

2 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.

2.1 Code source

2.1.1 Makefile

La version de base du Makefile qui nous était donnée, il est compliqué et exécuté sans aucune optimisation.

```
all: nbody.g
nbody.g: nbody.c
	gcc -g -mavx2 -fopt-info-all=nbody.gcc.optrpt $< -o $@ -lm -fopenmp
clean:
	rm -Rf *~ nbody.g  *.optrpt
```

2.1.2 Nbody.c

```
//
#include <omp.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

//
typedef float              f32;
typedef double             f64;
typedef unsigned long long u64;

//
typedef struct particle_s {

  f32 x, y, z;
  f32 vx, vy, vz;

} particle_t;

//
void init(particle_t *p, u64 n)
{
  for (u64 i = 0; i < n; i++)
    {
      //
```

```
        u64 r1 = (u64)rand();
        u64 r2 = (u64)rand();
        f32 sign = (r1 > r2) ? 1 : -1;

        //
        p[i].x = sign * (f32)rand() / (f32)RAND_MAX;
        p[i].y = (f32)rand() / (f32)RAND_MAX;
        p[i].z = sign * (f32)rand() / (f32)RAND_MAX;

        //
        p[i].vx = (f32)rand() / (f32)RAND_MAX;
        p[i].vy = sign * (f32)rand() / (f32)RAND_MAX;
        p[i].vz = (f32)rand() / (f32)RAND_MAX;
    }
}

//
void move_particles(particle_t *p, const f32 dt, u64 n)
{
  //
  const f32 softening = 1e-20;

  //
  for (u64 i = 0; i < n; i++)
    {
      //
      f32 fx = 0.0;
      f32 fy = 0.0;
      f32 fz = 0.0;

      //23 floating-point operations
      for (u64 j = 0; j < n; j++)
    {
      //Newton's law
      const f32 dx = p[j].x - p[i].x; //1
      const f32 dy = p[j].y - p[i].y; //2
      const f32 dz = p[j].z - p[i].z; //3
      const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening; //9
      const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0); //11

      //Net force
      fx += dx / d_3_over_2; //13
      fy += dy / d_3_over_2; //15
      fz += dz / d_3_over_2; //17
    }

      //
      p[i].vx += dt * fx; //19
      p[i].vy += dt * fy; //21
      p[i].vz += dt * fz; //23
    }

  //3 floating-point operations
  for (u64 i = 0; i < n; i++)
    {
```

```
81          p[i].x += dt * p[i].vx;
82          p[i].y += dt * p[i].vy;
83          p[i].z += dt * p[i].vz;
84        }
85    }
86
87    //
88    int main(int argc, char **argv)
89    {
90      //
91      const u64 n = (argc > 1) ? atoll(argv[1]) : 16384;
92      const u64 steps= 10;
93      const f32 dt = 0.01;
94
95      //
96      f64 rate = 0.0, drate = 0.0;
97
98      //Steps to skip for warm up
99      const u64 warmup = 3;
100
101     //
102     particle_t *p = malloc(sizeof(particle_t) * n);
103
104     //
105     init(p, n);
106
107     const u64 s = sizeof(particle_t) * n;
108
109     printf("\n\033[1mTotal memory size:\033[0m %llu B, %llu KiB, %llu MiB\
        n\n", s, s >> 10, s >> 20);
110
111     //
112     printf("\033[1m%5s %10s %10s %8s\033[0m\n", "Step", "Time, s", "
        Interact/s", "GFLOP/s"); fflush(stdout);
113
114     //
115     for (u64 i = 0; i < steps; i++)
116       {
117         //Measure
118         const f64 start = omp_get_wtime();
119
120         move_particles(p, dt, n);
121
122         const f64 end = omp_get_wtime();
123
124         //Number of interactions/iterations
125         const f32 h1 = (f32)(n) * (f32)(n - 1);
126
127         //GFLOPS
128         const f32 h2 = (23.0 * h1 + 3.0 * (f32)n) * 1e-9;
129
130         if (i >= warmup)
131       {
132         rate += h2 / (end - start);
133         drate += (h2 * h2) / ((end - start) * (end - start));
```

```
134         }
135
136           //
137           printf("%5llu %10.3e %10.3e %8.1f %s\n",
138              i,
139              (end - start),
140              h1 / (end - start),
141              h2 / (end - start),
142              (i < warmup) ? "*" : "");
143
144           fflush(stdout);
145         }
146
147       //
148       rate /= (f64)(steps - warmup);
149       drate = sqrt(drate / (f64)(steps - warmup) - (rate * rate));
150
151       printf("-------------------------------------------------------------------------------\n");
152       printf("\033[1m%s %4s \033[42m%10.1lf +- %.1lf GFLOP/s\033[0m\n",
153          "Average performance:", "", rate, drate);
154       printf("-------------------------------------------------------------------------------\n");
155
156       //
157       free(p);
158
159       //
160       return 0;
161   }
```

2.2 Déroulement du processur d'optimisation

2.2.1 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

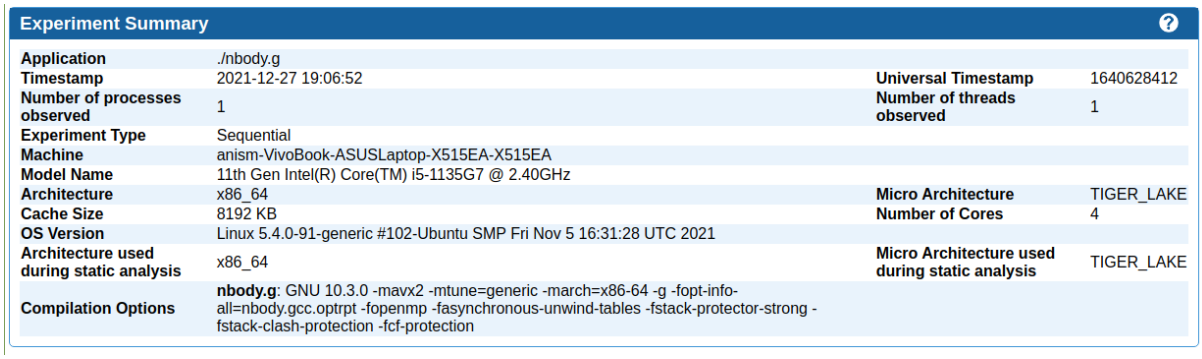
FIGURE 1 – Global Metrics.

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'a 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

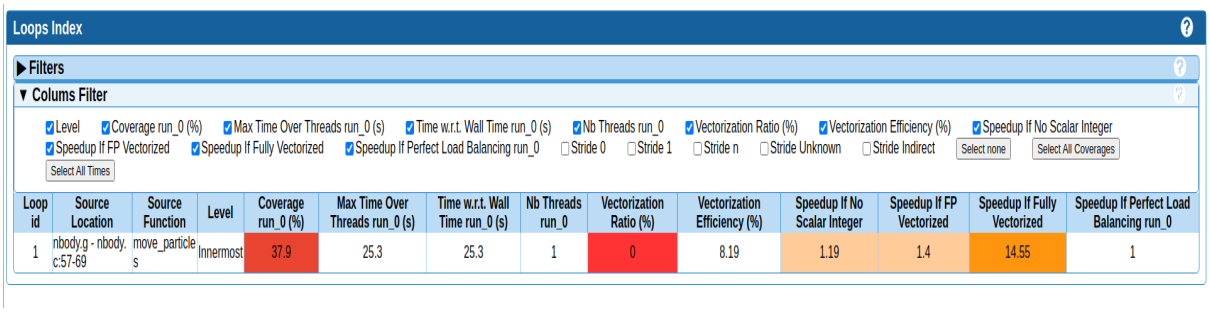
Application	./nbody.g		
Timestamp	2021-12-27 19:06:52	**Universal Timestamp**	1640628412
Number of processes observed	1	**Number of threads observed**	1
Experiment Type	Sequential		
Machine	anism-VivoBook-ASUSLaptop-X515EA-X515EA		
Model Name	11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz		
Architecture	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 -fopt-info-all=nbody.gcc.optrpt -fopenmp -fasynchronous-unwind-tables -fstack-protector-strong -fstack-clash-protection -fcf-protection		

FIGURE 2 – Experiment Summary.

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.



Loops Index

▶ Filters

▼ Colums 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
1	nbody.g - nbody.c:57-69	move_particles	Innermost	37.9	25.3	25.3	1	0	8.19	1.19	1.4	14.55	1

FIGURE 3 – Loops Index.

Comme on peut le voir, on a un tableau récapitulatif des boucles que nous devrons 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 devrons 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

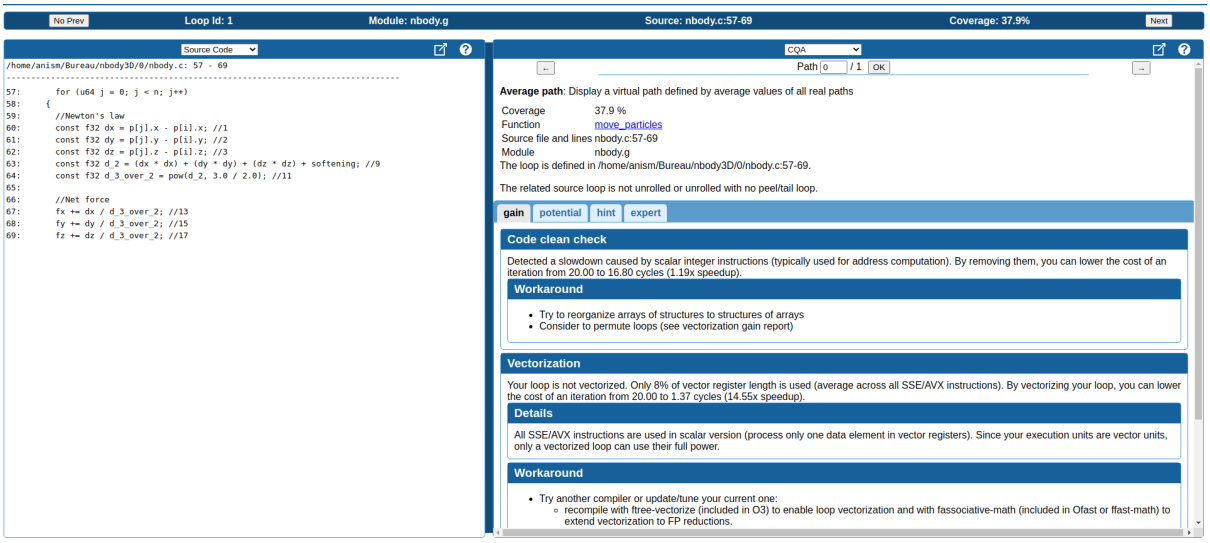


FIGURE 4 – Rapport CQA de la boucle N°1.

code source de la boucle sur la gauche et les améliorations à effectué dessus sur la droite. Nous allons réaliser chacune des modifications demandée lorsque cela est possible afin de pouvoir constater dans une analyse ultérieur 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.

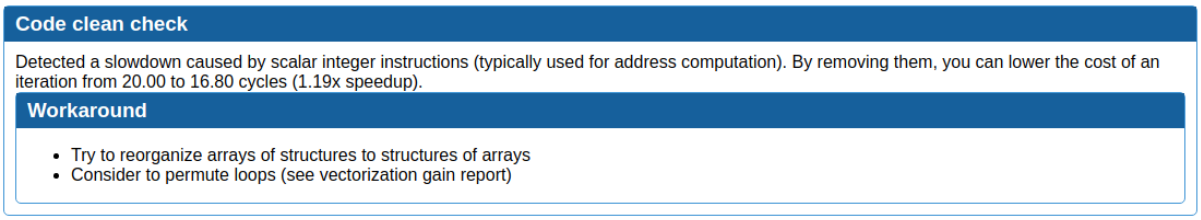


FIGURE 5 – Gain : Code clean check.

- **Gain : Vectorization**

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

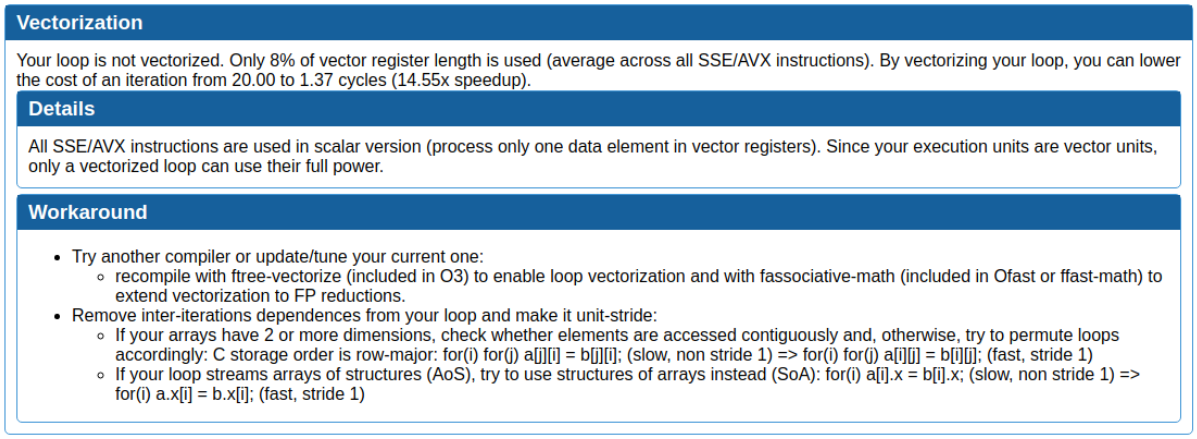


FIGURE 6 – Gain : Vectorization.

- **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 dans la mesure du possible et qu'on doit recompiler avec le flag **-march=tigerlake**

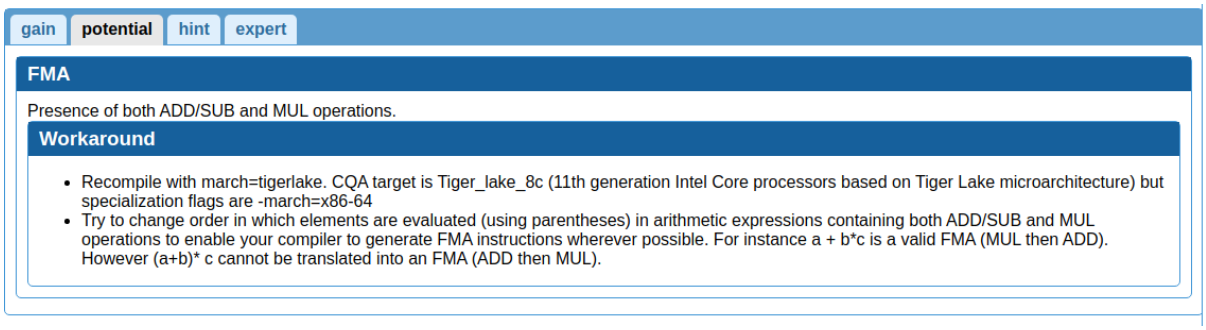


FIGURE 7 – Potential : FMA.

- **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.

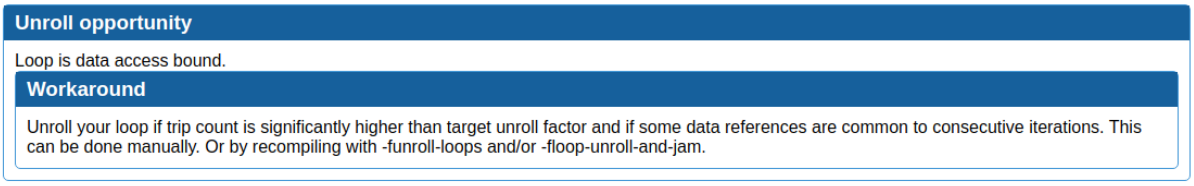


FIGURE 8 – Hint : Unroll opportunity.

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.

FIGURE 9 – Summary :Stylizer.

2.2.2 Version 1 corrigé de Nbody3D

Makefile corrigé

```
all: nbody.g nbody.g1

nbody.g: nbody.c
	gcc -g -mavx2 -fopt-info-all=nbody.gcc.optrpt $< -o $@ -lm -fopenmp

nbody.g1: nbody1.c
	gcc  -mavx2 -funroll-loops -march=tigerlake -finline-functions -
	fassociative-math -ftree-vectorize -Ofast -g -fopt-info-all=nbody.gcc
	.optrpt $< -o $@ -lm -fopenmp


clean:
	rm -Rf *~ nbody.g nbody.g1 *.optrpt
```

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

```

void init(particle_t *p, u64 n)
{
    p->x = malloc(sizeof(f32) * n);
    p->y = malloc(sizeof(f32) * n);
    p->z = malloc(sizeof(f32) * n);
    p->vx = malloc(sizeof(f32) * n);
    p->vy = malloc(sizeof(f32) * n);
    p->vz = malloc(sizeof(f32) * n);
  for (u64 i = 0; i < n; i++)
    {
      //
      u64 r1 = (u64)rand();
      u64 r2 = (u64)rand();
      f32 sign = (r1 > r2) ? 1 : -1;

      //
      p->x[i] = sign * (f32)rand() / (f32)RAND_MAX;
      p->y[i] = (f32)rand() / (f32)RAND_MAX;
      p->z[i] = sign * (f32)rand() / (f32)RAND_MAX;

      //
      p->vx[i] = (f32)rand() / (f32)RAND_MAX;
      p->vy[i] = sign * (f32)rand() / (f32)RAND_MAX;
      p->vz[i]= (f32)rand() / (f32)RAND_MAX;
    }
}
```

Changement en l'ordre de SOA de la fonction move_particles

```
void move_particles(particle_t *p, const f32 dt, u64 n)
{
  //
  const f32 softening = 1e-20;

  //
  for (u64 i = 0; i < n; i++)
    {
      //
      f32 fx = 0.0;
      f32 fy = 0.0;
      f32 fz = 0.0;

      //23 floating-point operations
      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; //9
    const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0); //11

    //Net force
    fx += dx / d_3_over_2; //13
    fy += dy / d_3_over_2; //15
    fz += dz / d_3_over_2; //17
  }

      //
      p->vx[i] += dt * fx; //19
      p->vy[i] += dt * fy; //21
      p->vz[i] += dt * fz; //23
    }

  //3 floating-point operations
  for (u64 i = 0; i < n; i+=4)
    {
      p->x[i] += dt * p->vx[i];
      p->y[i] += dt * p->vy[i];
      p->z[i] += dt * p->vz[i];

      p->x[i+1] += dt * p->vx[i+1];
      p->y[i+1] += dt * p->vy[i+1];
      p->z[i+1] += dt * p->vz[i+1];


      p->x[i+2] += dt * p->vx[i+2];
      p->y[i+2] += dt * p->vy[i+2];
      p->z[i+2] += dt * p->vz[i+2];


```

```
53          p->x[i+3] += dt * p->vx[i+3];
54          p->y[i+3] += dt * p->vy[i+3];
55          p->z[i+3] += dt * p->vz[i+3];
56      }
57  }
```

Changement de l'enroulement de la boucle

```
1   for (u64 i = 0; i < n; i+=4)
2       {
3         p->x[i] += dt * p->vx[i];
4         p->y[i] += dt * p->vy[i];
5         p->z[i] += dt * p->vz[i];
6
7         p->x[i+1] += dt * p->vx[i+1];
8         p->y[i+1] += dt * p->vy[i+1];
9         p->z[i+1] += dt * p->vz[i+1];
10
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
17        p->x[i+3] += dt * p->vx[i+3];
18        p->y[i+3] += dt * p->vy[i+3];
19        p->z[i+3] += dt * p->vz[i+3];
20      }
```

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 éxecution du programme, on remarque notre programme est améliorer 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 est au plus efficace possible : gain de temps d'exécution constaté par rapport à toutes la version de base précédente. Nous avons donc ainsi atteint le meilleur code possible avec les meilleurs options d'optimisation possible pour le Nbody3D.

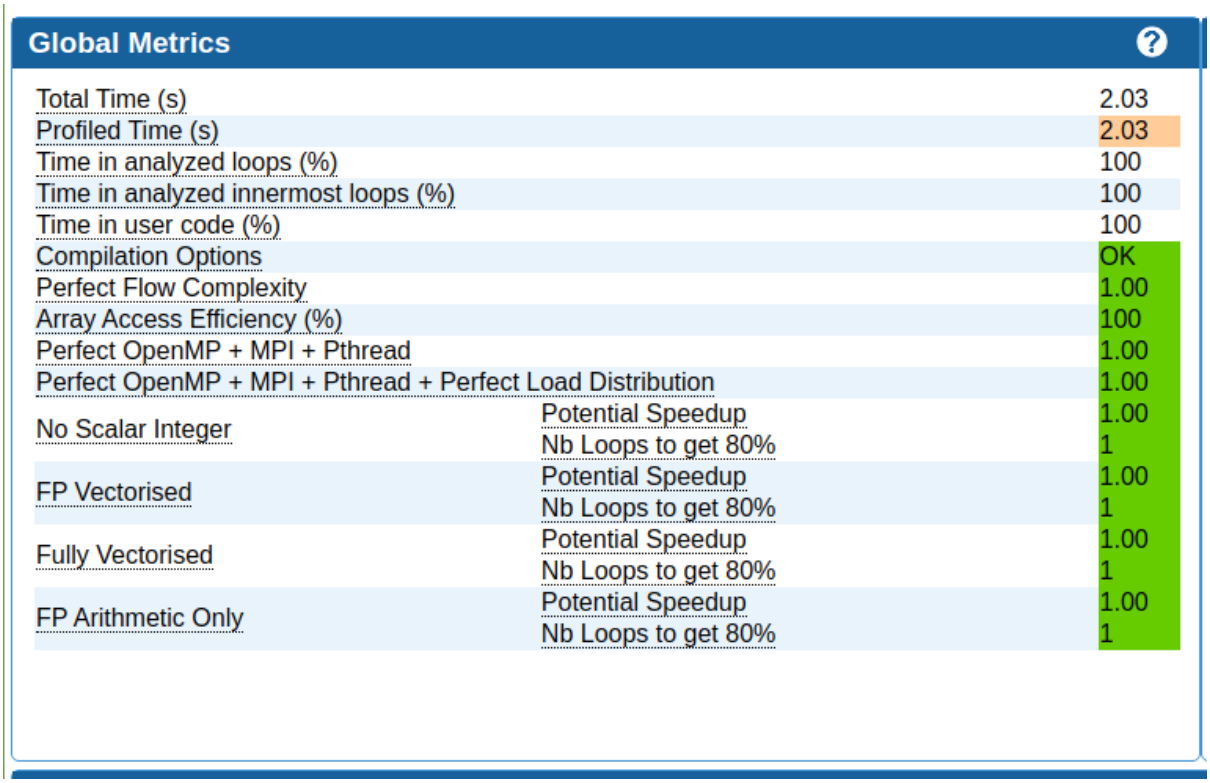


FIGURE 10 – Global Metrics de la version optimisé.

2. **La page Loops**

   On voit que les parameètres sont très suffisant, il manque la Vectorization Efficiency, on consulte son rapport CQA afin de voir les modifications qu'il faut apporter.

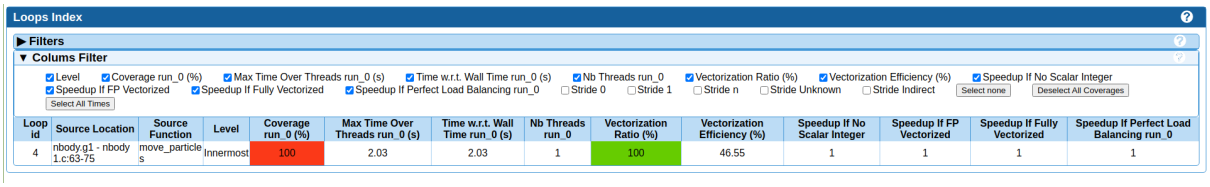


FIGURE 11 – Loops Index de la version 1 optimisé.

3. **La page Summary**

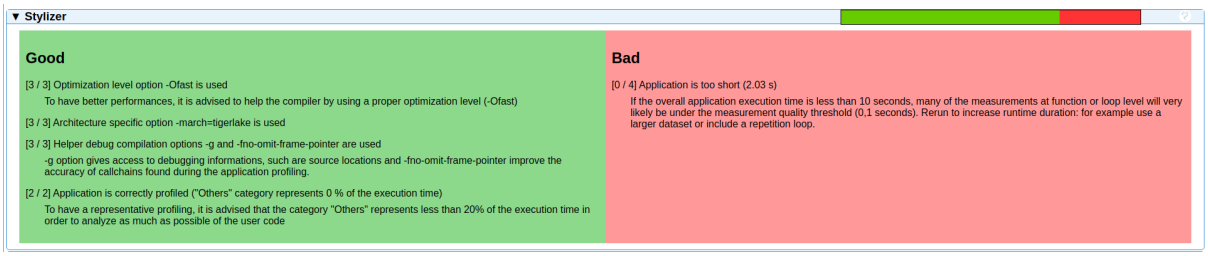


FIGURE 12 – Summary : Stylizer de la version 1 optimisé.

3. **Rapport CQA**

- **Gain :** On voit que notre boucle est vectorisé, mais que 46% de la longueur du registre.

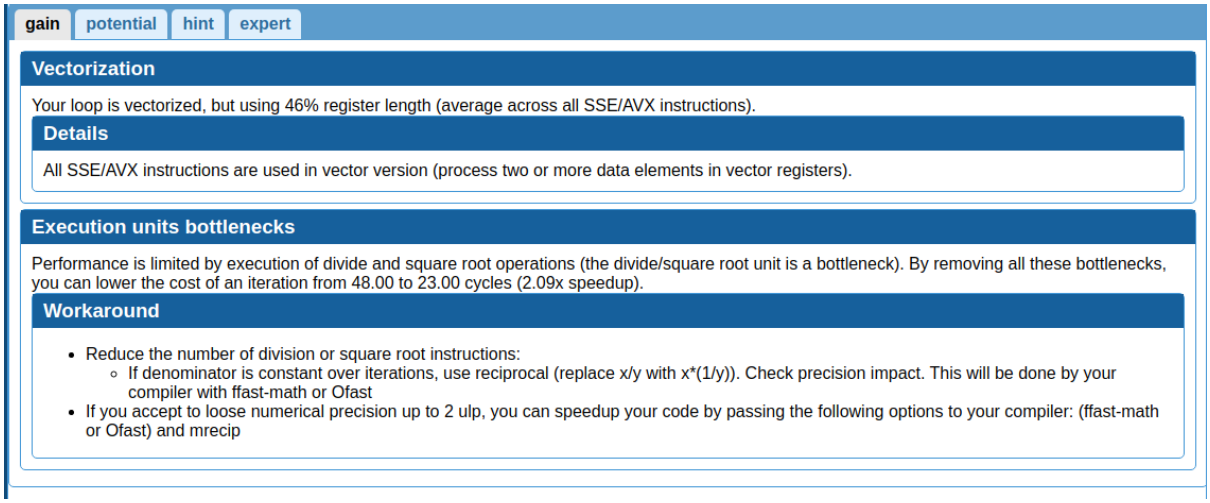


FIGURE 13 – Gain de la version 1 optimisé.

- **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 applicons donc la suggestion d'utiliser la fonction **posix_memalign** .

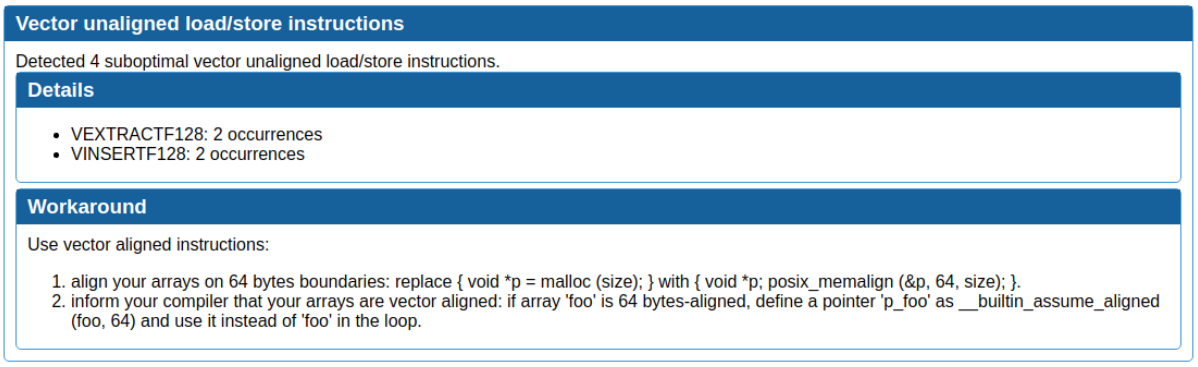


FIGURE 14 – Vector unaligned load/store instructions de la version 1 optimisé.