1 Nbody2D

Makefile

Comme on peut le voir sur le Makefile suivant, notre programme est compilé sans aucune optimisation afin de commencer notre analyse depuis le début.

```
nbody0: nbody0.c
	gcc -g -funroll-loops -finline-functions -ftree-vectorize $< -o $@ -lm
```

FIGURE 1 – Makefile

1.1 Sortie de Maqao

Une fois MAQAO lancer avec les flags de base lui permettant à faire son analyse. On a des fichier htmls en sortie on exploite alors les informations mises a disposition afin d'optimiser notre programme :

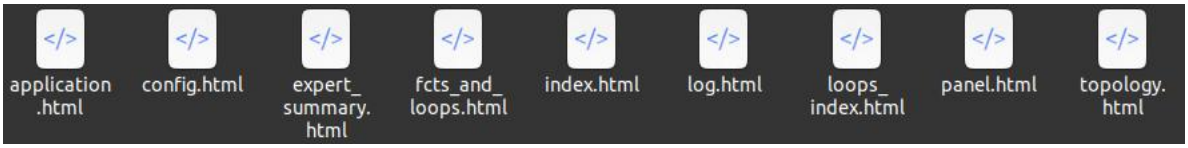


FIGURE 2 – Sortie de l'execution de MAQAO

1.1.1 Index

On commence notre analyse par la page d'accueil du rapport. Sur cette page, on retrouve les informations sur le temps d'execution et si il est possible d'optimiser notre programme ou pas et les infos relative a la machine d'execution de programme.

Global Metrics

Total Time (s)		23.96
Profiled Time (s)		23.96
Time in analyzed loops (%)		19.5
Time in analyzed innermost loops (%)		19.4
Time in user code (%)		48.5
Compilation Options		**nbody0**: -O2, -O3 or -Ofast is missing. -march=(target) is missing.
Perfect Flow Complexity		1.02
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	1.11
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.11
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.20
	Nb Loops to get 80%	1
FP Arithmetic Only	Potential Speedup	1.17
	Nb Loops to get 80%	1

FIGURE 3 – Global Metrics

En observant les métriques globales du binaire analysé, on constate que celui-ci a été compilé sans flags d'optimisation ni de flags de spécification d'architecture. De plus, on voit que d'une part, nos accès mémoire sont efficaces à 83.3% (la valeur est bonne mais pourrait être amélioré) et des speed-up peuvent-être otbtenus si le programme est vectorisé à la compilation.

A cette étape, nous allons prendre en compte la suggestion des flags -O2, -O3 or -Ofast, -march=(target) et pour le prochain binaire à produire.

1.1.2 Experiment Summary

Experiment Summary

Application	./nbody0		
Timestamp	2021-12-28 12:48:01	**Universal Timestamp**	1640692081
Number of processes observed	1	**Number of threads observed**	1
Experiment Type	Sequential		
Machine	madjid-Inspiron-3543		
Model Name	Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz		
Architecture	x86_64	**Micro Architecture**	BROADWELL
Cache Size	3072 KB	**Number of Cores**	2
OS Version	Linux 5.11.0-43-generic #47~20.04.2-Ubuntu SMP Mon Dec 13 11:06:56 UTC 2021		
Architecture used during static analysis	x86_64	**Micro Architecture used during static analysis**	BROADWELL
Compilation Options	**nbody0**: 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		

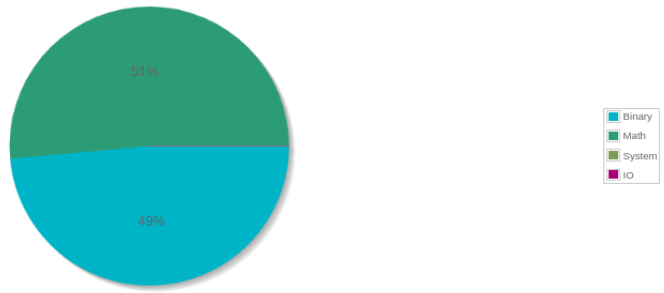
FIGURE 4 – Experiment-Summary

Experiment Summary nous donne les informations lien à la machine d'execution et ce que le compilateur a ajouter a notre place.

1.2 Application

La partie application nous donne des détails sur comment le temps a etait passé entre plusieurs categories (System, Binary, Math, IO etc..) et ou notre application passe beaucoup de temps.

Detailed Application Categorization



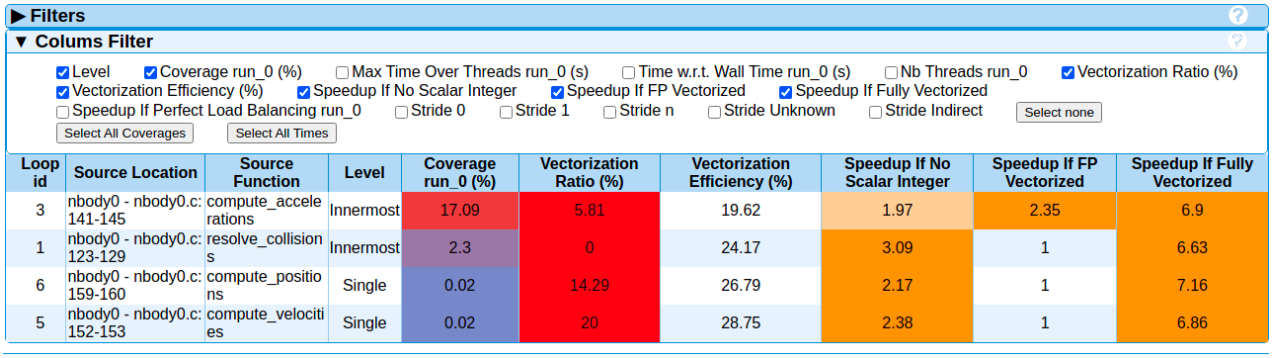
ID	Time(s)	Binary(%)	MPI(%)	OMP(%)	Math(%)	System(%)	Pthread(%)	IO(%)	String(%)	Memory(%)	Others(%)
▼ run_0	23.96	48.55	0	0	51.41	0.02	0	0.02	0	0	0
▼ Node madjid-Inspiron-3543	23.96	48.55	0	0	51.41	0.02	0	0.02	0	0	0
▼ Process 4570	23.96	48.55	0	0	51.41	0.02	0	0.02	0	0	0
○ Thread 4570	23.96	48.55	0	0	51.41	0.02	0	0.02	0	0	0

FIGURE 5 – Applications

Dans notre cas notre programme passe 51.41% de son temps dans les methodes mathématiques et 48.55% dans binary(code utilisateur).

1.3 Loops

Ici dans cette section on retrouve toutes les boucles qui prennes de temps a s'éxecuter par ordre et les détails sur la convergence la vectorizations etc...



▶ 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

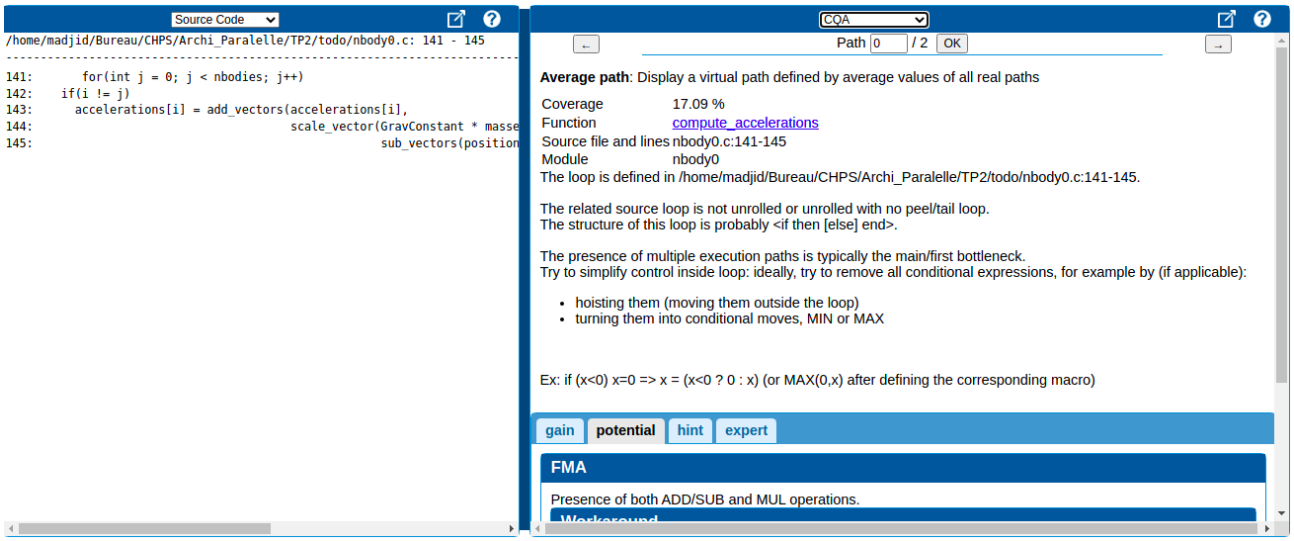
Loop id	Source Location	Source Function	Level	Coverage run_0 (%)	Vectorization Ratio (%)	Vectorization Efficiency (%)	Speedup If No Scalar Integer	Speedup If FP Vectorized	Speedup If Fully Vectorized
3	nbody0 - nbody0.c: 141-145	compute_accelerations	Innermost	17.09	5.81	19.62	1.97	2.35	6.9
1	nbody0 - nbody0.c: 123-129	resolve_collisions	Innermost	2.3	0	24.17	3.09	1	6.63
6	nbody0 - nbody0.c: 159-160	compute_positions	Single	0.02	14.29	26.79	2.17	1	7.16
5	nbody0 - nbody0.c: 152-153	compute_velocities	Single	0.02	20	28.75	2.38	1	6.86

FIGURE 6 – Loops

On remarque qu'une boucle à une couverture de 17% on cliquant sur elle on affiche son rapport **CQA**.

1.3.1 Rapport CQA

Le rapport CQA il nous indique quelle boue de boucle consomme le plus et se présente comme la montre la figure suivante :



Source Code

/home/madjid/Bureau/CHPS/Archi_Paralelle/TP2/todo/nbody0.c: 141 - 145

```
141:       for(int j = 0; j < nbodies; j++)
142:     if(i != j)
143:       accelerations[i] = add_vectors(accelerations[i],
144:                                      scale_vector(GravConstant * masse
145:                                                   sub_vectors(position
```

CQA

← Path 0 / 2 OK →

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

Coverage 17.09 %
Function compute_accelerations
Source file and lines nbody0.c:141-145
Module nbody0
The loop is defined in /home/madjid/Bureau/CHPS/Archi_Paralelle/TP2/todo/nbody0.c:141-145.

The related source loop is not unrolled or unrolled with no peel/tail loop.
The structure of this loop is probably <if then [else] end>.

The presence of multiple execution paths is typically the main/first bottleneck.
Try to simplify control inside loop: ideally, try to remove all conditional expressions, for example by (if applicable):

- hoisting them (moving them outside the loop)
- turning them into conditional moves, MIN or MAX

Ex: if (x<0) x=0 => x = (x<0 ? 0 : x) (or MAX(0,x) after defining the corresponding macro)

gain potential hint expert

FMA

Presence of both ADD/SUB and MUL operations.

Workaround

FIGURE 7 – Rapport CQA

Détails de rapport CQA

Gain

Dans cette partie MAQAO nous fais une estimations de gain de temps on vectorisant notre boucle et on donnant solution de contournement en changent la structure de Arrays of Structure (AoS) à Structure of Arrays (SoA).

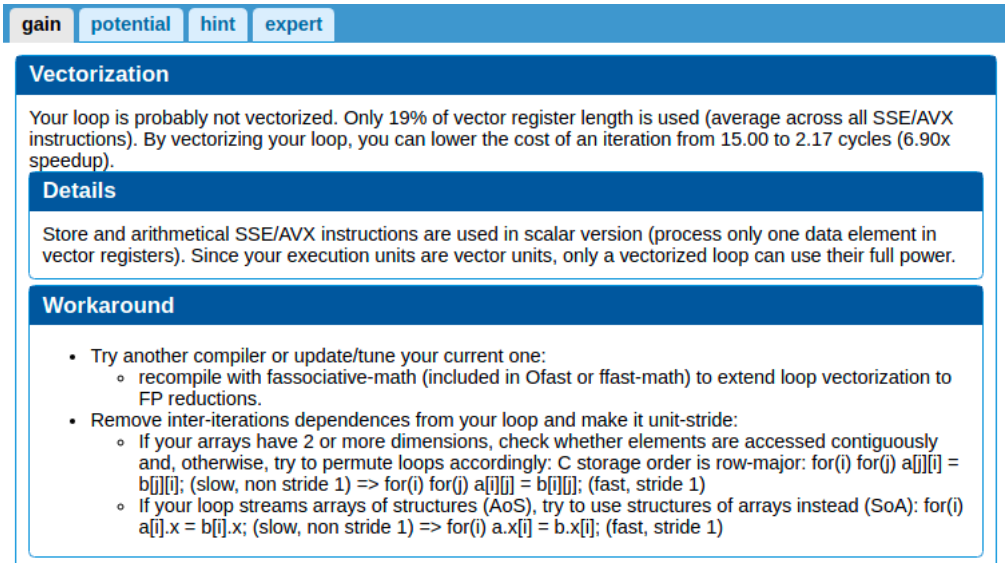


FIGURE 8 – Gain

Potential

MAQAO nous recommande de recompilé avec march=broadwell vu que il a detecter notre architecture est nous proposes la meilleur target possible.

MAQAO détecter aussi la présence d'ADD/SUB et MUL et il nous recommande de changer de synthaxe et passé de a + b*c est un FMA valide (MUL puis ADD). Cependant (a+b)* c ne peut pas être traduit en FMA (ADD puis MUL).

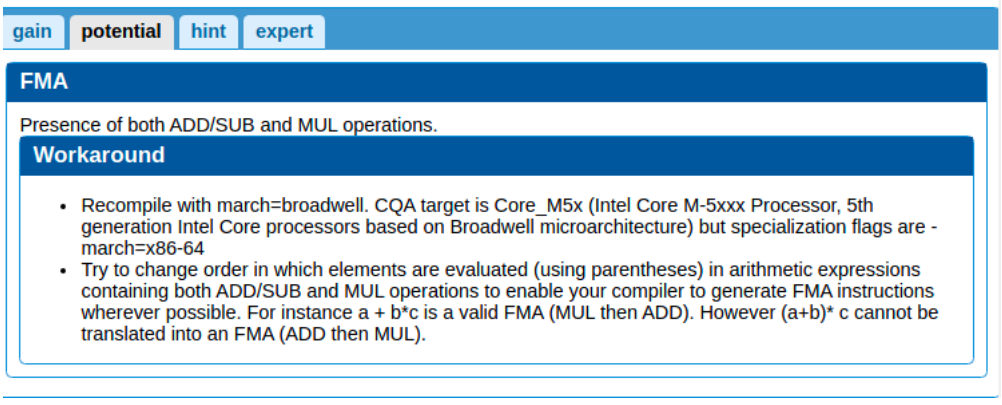


FIGURE 9 – potential

Hint

Dans cette partie MAQAO nous fais la correspondance entre nous differentes boucles (dans le code source) et la boucle binaire dans notre cas ils nous donne combien d'opérations arithmétiques notre boucle éxecute.

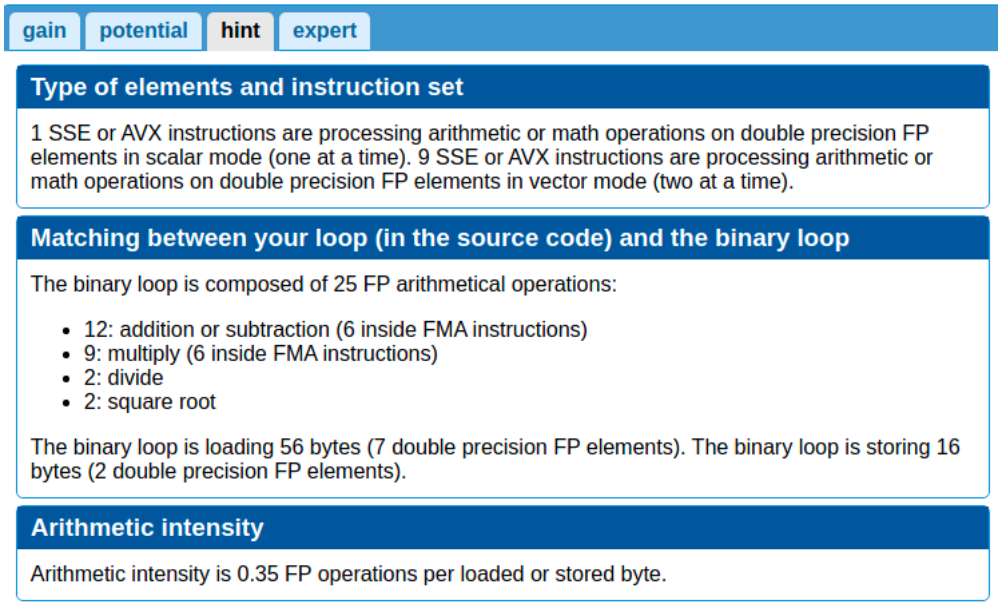


FIGURE 10 – Hint

2 Optimiser

Makefile

Cette fois on prend on compte les suggestion de MAQAO on changent la ligne de compilation on ajoutant -Ofast -funroll-loops -march=broadwell.

```
nbody0: nbody0.c
	gcc -g -Ofast -funroll-loops -march=broadwell -finline-functions -ftree-vectorize $< -o $@ -lm
```

FIGURE 11 – Makefile modifier

2.1 Global Metrics

Maintenant on remaque que on changent la compilation de Makefile on a ganger on temps et on vectorization et toutes est devenu verts.

Global Metrics

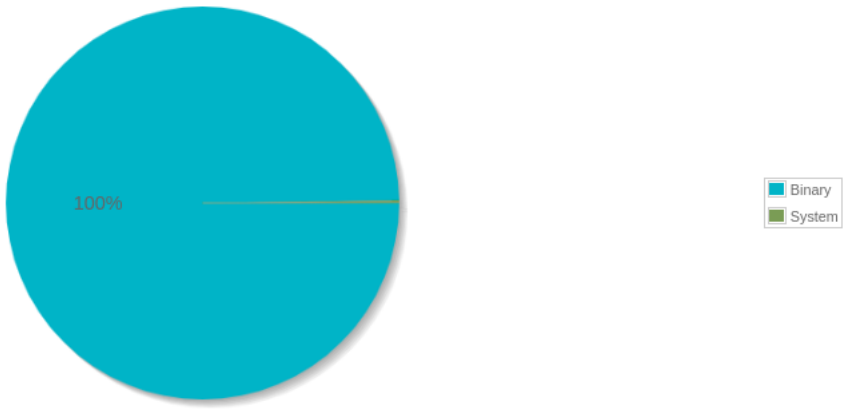
Total Time (s)		2.44
Profiled Time (s)		2.44
Time in analyzed loops (%)		99.8
Time in analyzed innermost loops (%)		98.1
Time in user code (%)		99.8
Compilation Options		OK
Perfect Flow Complexity		1.06
Array Access Efficiency (%)		83.3
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.01
	Nb Loops to get 80%	2
FP Vectorised	Potential Speedup	1.00
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.07
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	1.05
	Nb Loops to get 80%	1

FIGURE 12 – Global Metrics optimiser

2.2 Application

Meme dans l'application on remarque maintenant que le code s'éxecute à 99.79% dans le code utilisateur.

Detailed Application Categorization



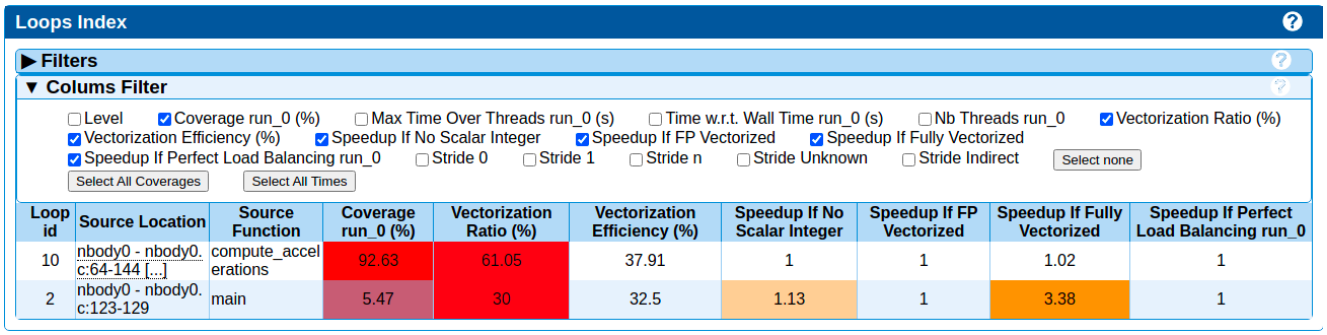
ID	Time(s)	Binary(%)	MPI(%)	OMP(%)	Math(%)	System(%)	Pthread(%)	IO(%)	String(%)	Memory(%)	Others(%)
▼ run_0	2.44	99.79	0	0	0	0.21	0	0	0	0	0
▼ Node madjid-Inspiron-3543	2.44	99.79	0	0	0	0.21	0	0	0	0	0
▼ Process 5851	2.44	99.79	0	0	0	0.21	0	0	0	0	0
○ Thread 5851	2.44	99.79	0	0	0	0.21	0	0	0	0	0

FIGURE 13 – Application optimiser

2.3 Loops

2.3.1 Loops Index

Le code converge a 93% et il s'est vectoriser a 61.05%.



Loop id	Source Location	Source Function	Coverage 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
10	nbody0 - nbody0.c:64-144 [...]	compute_accelerations	92.63	61.05	37.91	1	1	1.02	1
2	nbody0 - nbody0.c:123-129	main	5.47	30	32.5	1.13	1	3.38	1

FIGURE 14 – Hint