



# Rapport de projet

Mesures de performance  
N-body simulation



**Professeurs:** **M. Mohammed-Salah IBNAMAR**  
**M. Hugo BOLLORE**  
**M. William JALBY**

# Table des matières

Introduction	3
Optimisations	4
Protocole de mesure	7
Résultats	8
Bilan	12

# Introduction

La simulation N-body offre un aperçu fascinant de la manière dont la modélisation peut nous aider à comprendre des phénomènes complexes. Et cela de l'échelle subatomique aux corps célestes.

Cependant, la précision et l'efficacité de ces simulations dépendent en grande partie de la qualité de l'implémentation logicielle.

De plus, les limites du matériel informatique restent un défi en termes de puissance de calcul et de consommation d'énergie.

C'est pourquoi, quels que soient les moyens dont on dispose, l'optimisation du code est une compétence précieuse lorsqu'on souhaite effectuer une telle simulation.

## ***Mais comment améliorer la qualité du code d'une simulation N-body ?***

Nous commencerons par voir les différentes techniques d'optimisations utilisées dans ce projet, puis nous verrons le protocole de mesure. Enfin, nous analyserons les résultats obtenus.

# Optimisations

```
typedef struct particle_s_SOA
{
    f32 *x, *y, *z;
    f32 *vx, *vy, *vz;
} particle_t_SOA;
```

Structure of Arrays (SOA)

La structure initiale du code repose sur un modèle Array Of Structures (AOS).

J'ai implémenté une SOA car c'est généralement plus rapide que l'AOS dans les cas où il est nécessaire d'accéder à tous les éléments d'un même champ de données de manière séquentielle.

Cela est dû au fait que la SOA stocke tous les éléments d'un même champ dans un tableau séparé, ce qui facilite l'accès séquentiel à ces éléments.

## Memory Alignment

```
p.x = aligned_alloc(sizeof(f32), n * sizeof(f32));
```

Lorsque les données sont correctement alignées dans la mémoire, elles peuvent être traitées plus efficacement par le processeur.

En effet, ce dernier peut ainsi accéder à plusieurs octets à la fois plutôt que d'accéder à des octets individuels.

Cela peut entraîner une amélioration des performances, car le processeur peut traiter les données plus rapidement.

## Parallélisation

```
#pragma omp parallel for schedule(static, (n / omp_get_num_threads()))
```

Les directives d'openMP permettent de paralléliser des parties de code. L'exécution est ainsi répartie entre plusieurs threads qui s'exécute en même temps.

Après différents tests, j'ai trouvé que paralléliser la boucle la plus externe de move\_particles() et la petite boucle en fin de fonction permettait plus de gain.

## Vectorisation

```
#pragma omp simd simdlen(8)
```

La vectorisation permet de traiter simultanément plusieurs éléments de données en effectuant des calculs en parallèle (Single Instruction Multiple Data), ce qui peut améliorer significativement les performances des programmes.

## Remplacement des instructions coûteuses

Les divisions sont considérées comme des opérations coûteuses pour les processeurs car elles sont beaucoup plus complexes que les opérations arithmétiques simples telles que l'addition, la soustraction ou la multiplication.

Les processeurs modernes sont capables d'exécuter de nombreuses instructions en parallèle, ce qui permet d'optimiser les performances. Cependant, pour les divisions, cela n'est pas possible car elles nécessitent plusieurs cycles d'horloge pour être exécutées, ce qui ralentit le temps de traitement.

J'ai donc remplacé les divisions du code initial.

La première instruction couteuse qu'on remarque est  $\text{pow}(\text{d\_2}, 3.0 / 2.0)$ .

Or,  $\text{pow}(\text{d\_2}, 3.0 / 2.0) = \text{pow}(\text{d\_2}, (1.0 / 2.0) + 1)$

$$= d_2 * \text{sqrtf}(d_2) \quad (x \text{ puissance } 1/2 \text{ étant égale à racine carrée}(x))$$

Par la suite, j'ai également remarqué, au niveau des calculs de fx, fy et fz, que 3 divisions sont effectuées avec le même diviseur:  $d_2 * \text{sqrtf}(d_2)$ .

La multiplication étant moins coûteuse en cycles que la division, il est plus rapide de d'abord calculer l'inverse de ce diviseur (1 division) puis de faire 3 multiplications pour les calculs des forces.

## Fast Inverse Square Root Algorithm

f32 invSqrt(f32 number)

L'instruction  $1/(d_2 * \text{sqrtf}(d_2))$  reste toujours coûteuse. C'est pourquoi j'ai effectué des recherches sur stackoverflow pour voir je pouvais trouver un moyen de remplacer la racine carrée, et je suis tombé sur ce lien: <https://stackoverflow.com/questions/12923657/is-there-a-fast-c-or-c-standard-library-function-for-double-precision-inverse>

Il contient une fonction qui calcule de manière approximative mais rapide l'inverse de la racine carrée d'un nombre. J'ai pu facilement l'appliquer dans mon cas. En effet:

$$1/(d_2 * \text{sqrtf}(d_2)) = (1/\text{sqrtf}(d_2)) * (1/d_2) = \text{invsqrt}(d_2) * (1/d_2)$$

## Unrolling

J'ai choisi de dérouler la boucle interne ainsi que la petite boucle en fin de fonction.

J'avais en tête de limiter le nombre de tests effectués dans la ligne `for(...)` qui déterminent si la boucle s'arrête ou non.

Mais aussi de permettre au compilateur de potentiellement faire plus d'optimisations en lui fournissant davantage de variables "const" dont il peut prédire la valeur.

Il est à noter que j'ai effectué le déroulage sans l'utilisation de directives OpenMP.

# Protocole de mesure

## Caractéristiques du PC:

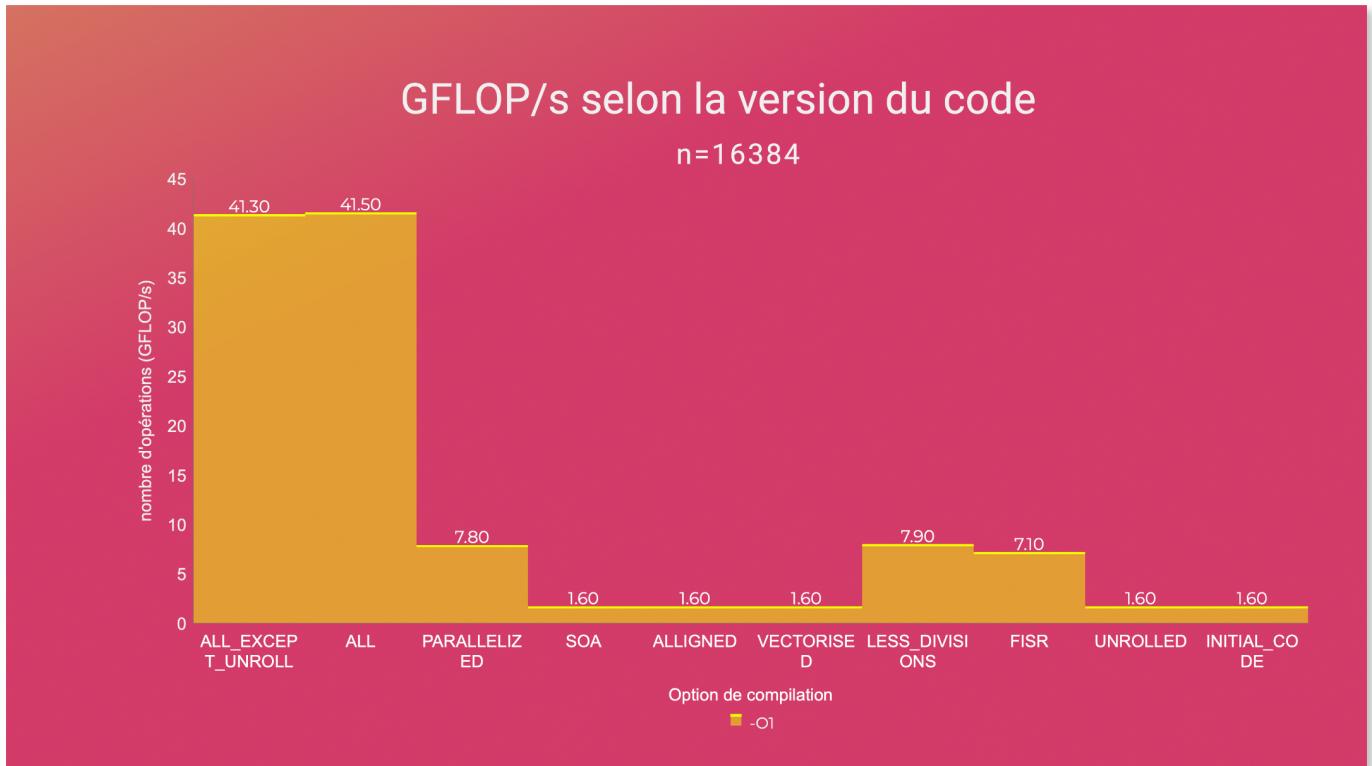
HWINFO64 v7.30-4870 @ MSI MS-7C94 - Résumé du système																		
CPU	AMD Ryzen 5 5600G			7 nm														
Pas à pas	CZN-A0			TDP 65 W														
Nom de ...	Cezanne			MCU A50000D														
OPN	100-000000252																	
CPU #0	Plate-forme			AM4														
Noyaux	6 / 12	Cache L1	6x32 + 6x32	L2 6x512 L3 16M														
Caractéristiques																		
MMX	3DNow!	3DNow!-2	SSE	SSE-2	SSE-3	SSE-3												
SSE4A	SSE4.1	SSE4.2	AVX	AVX2	AVX-512													
BMI2	ABM	TBM	FMA	ADX	XOP	AMX												
DEP	AMD-V	SMX	SMEP	SMAP	TSX	MPX												
EM64T	ELST	TM1	TM2	HTT	CPB	SST												
AES-NI	RDRAND	RDSEED	SHA	SGX	SME													
Point de fonctionnement	Horloge	iplicateur	Bus	VID														
Horloge minimale	400.0 MHz	x4.00	100.0 MHz	-														
Horloge de base	3900.0 MHz	x39.00	100.0 MHz	-														
Boost Max	4450.0 MHz	x44.50	100.0 MHz	-														
PBO Max	4450.0 MHz	x44.50	100.0 MHz	-														
Horloge Active Moyenne	4387.9 MHz	x43.96	99.8 MHz	1.4354 V														
Horloge Effic. Moyenne	245.3 MHz	x2.46	-	-														
CPU #0 - Horloge active																		
Core	Horloge	MHz	iplicateur															
0	4392	x4.00																
1	4392	x4.00																
2	4392	x4.00																
3	4392	x4.00																
4	4392	x4.00																
5	4367	x43.75																
Carte mère																		
MSI MAG B550 MORTAR WIFI (MS-7C94)			GPU			GPU												
Jeu de puces	AMD B550 (Promontory PROM19 C)			MSI RTX 3080 Ventus 3 OC LHR			GPU											
Date du BIOS	08/17/2022			NVIDIA GeForce RTX 3080			GPU											
Version	1.00			GA102-202 (LHR)			GPU											
PCIe v4.0 x16 (16.0 GT/s) @ x16 (8.0 GT/s)			PCIe v4.0 x16 (16.0 GT/s) @ x16 (8.0 GT/s)			GPU												
GPU #0	10 GB			GDDR6 SDRAM			GPU											
ROPs / TMUs	96 / 272			320-bit			GPU											
Horloges actuelles (MHz)	SH/RT/TC			8704 / 68 / 272			GPU											
GPU	1740.0			Mémoire			Vidéo											
Système opérateur																		
UEFI Boot	Secure Boot			TPM			HvCI											
Microsoft Windows 11 Professional (x64) Build 22621.1105																		
Disques durs																		
Interface	Modèle [Capacité]																	
NVMe 4x 8.0 GT/s	Force MP510 [480 GB]																	
NVMe 4x 16.0 GT/s	KINGSTON SNV2S2000G [2 TB]																	
CPU #0 - Horloge actives																		
Core	Horloge	MHz	iplicateur															
0	4392	x4.00																
1	4392	x4.00																
2	4392	x4.00																
3	4392	x4.00																
4	4392	x4.00																
5	4367	x43.75																
Carte mère																		
233'3	8	8	8	I8	52	-	I'50											
000'1	0	0	0	55	31	-	I'50											
800'0	11	11	11	55	38	-	I'50											
033'3	13	13	13	31	44	-	I'50											
100'1	12	12	12	30	20	-	I'50											
000'1	1	0	0	10	52	XWb	I'32											
800'0	8	10	10	10	50	XWb	I'32											
033'3	10	15	15	53	34	XWb	I'32											
100'1	13	13	13	50	39	XWb	I'32											
000'1	11	14	14	50	39	XWb	I'32											
100'1	13	13	13	50	39	XWb	I'32											

Le ryzen 5 5600G dispose de 6 coeurs physiques et 6 coeurs logiques. C'est pourquoi j'ai utilisé un OMP\_NUM\_THREADS=12. Chaque mesure a été effectuée avec le minimum de processus actifs sur mon ordinateur.

Enfin, la compilation a été faite avec GCC car il me donnait les meilleurs résultats. AOCC me donne des warnings qu'il ne peut pas effectuer certaines optimisations (notamment des loops not vectorized). Le maximum atteint sur AOCC a été 65.4 (+0.8) GFLOP/s.

Les nombres de GFLOPS/s qui figurent dans les diagrammes suivants sont chacun un résultat d'une moyenne de 5 mesures.

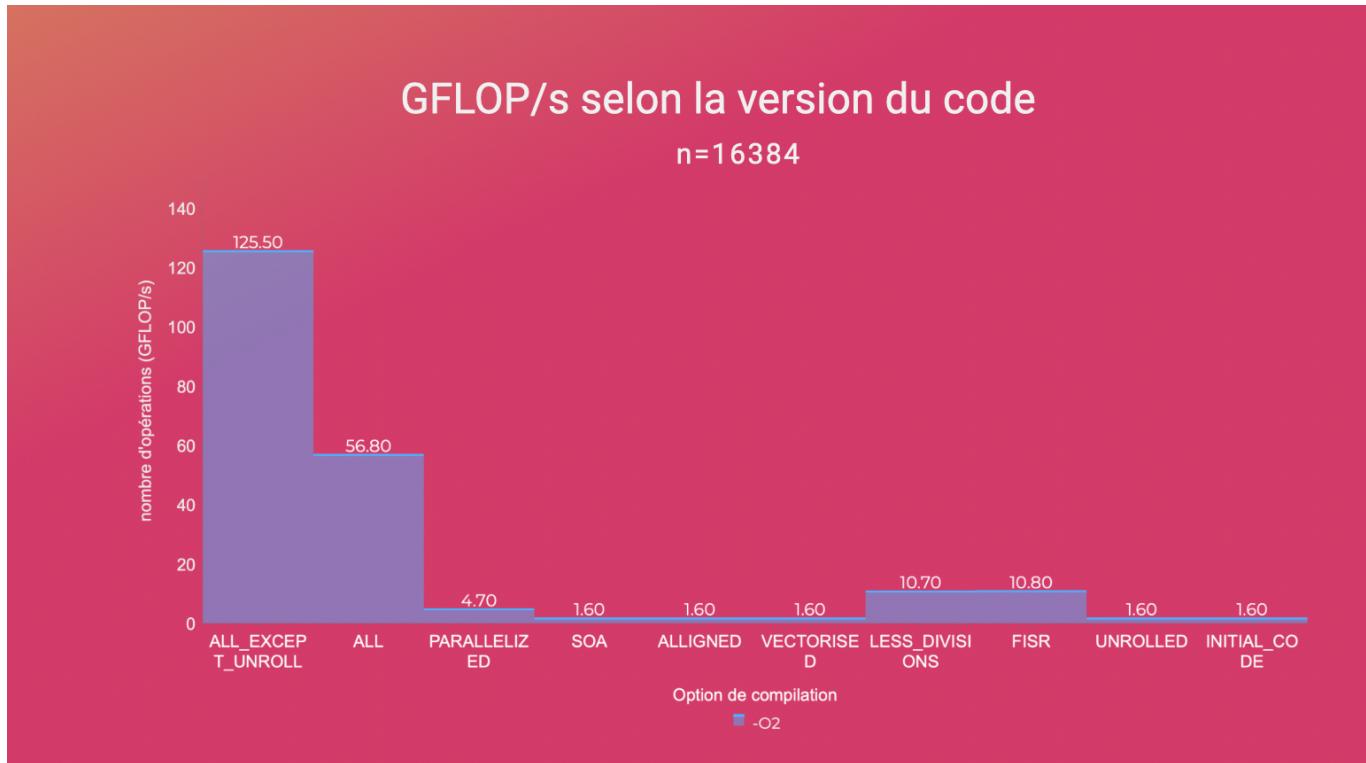
# Résultats



Lors de l'utilisation de l'option `-O1`, le nombre de GFLOP/s pour le code initial a été mesuré à 1,6 (+-0).

Bien que plusieurs optimisations telles que SOA, ALIGNED, VECTORISED et UNROLLED n'aient pas amélioré la vitesse d'exécution du code, les versions FISR (Fast Inverse Square Root), LESS\_DIVISIONS et PARALLELIZED ont montré une augmentation d'environ 5,5 à 6,3 GFLOP/s par rapport au code initial.

Par ailleurs, lorsque toutes les optimisations ont été cumulées, une augmentation encore plus significative a été observée, avec en moyenne de 41.4 GFLOP/s pour les versions ALL et ALL\_EXCEPT\_UNROLL.



Lorsque l'option de compilation `-O2` a été utilisée, les optimisations qui avaient précédemment échoué n'ont montré aucun changement.

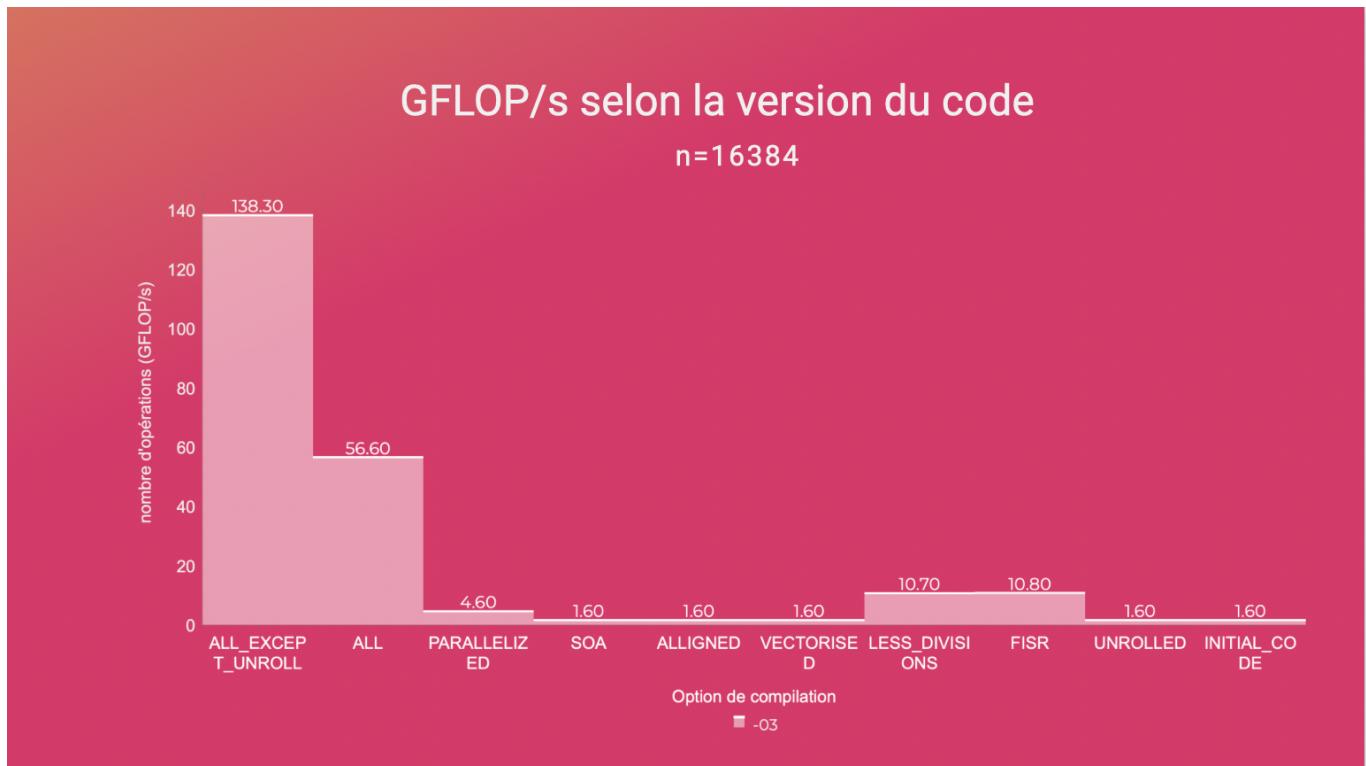
La version **PARALLELIZED** est la seule qui a subit une légère diminution de son temps d'exécution d'environ 3 GFLOP/s.

En revanche, une légère augmentation (d'environ 3 GFLOP/s) a été observée pour les versions **FISR** et **LESS\_DIVISIONS**.

La version **ALL** subit une nette amélioration d'environ 15 GFLOP/s tandis que la version **ALL\_EXCEPT\_UNROLLED** voit son nombre de GFLOP/s tripler, passant de 41.3 à 125.5 GFLOP/s. On peut déduire de cette comparaison que l'unrolling ralentit la vitesse d'exécution du code dans ce cas précis.

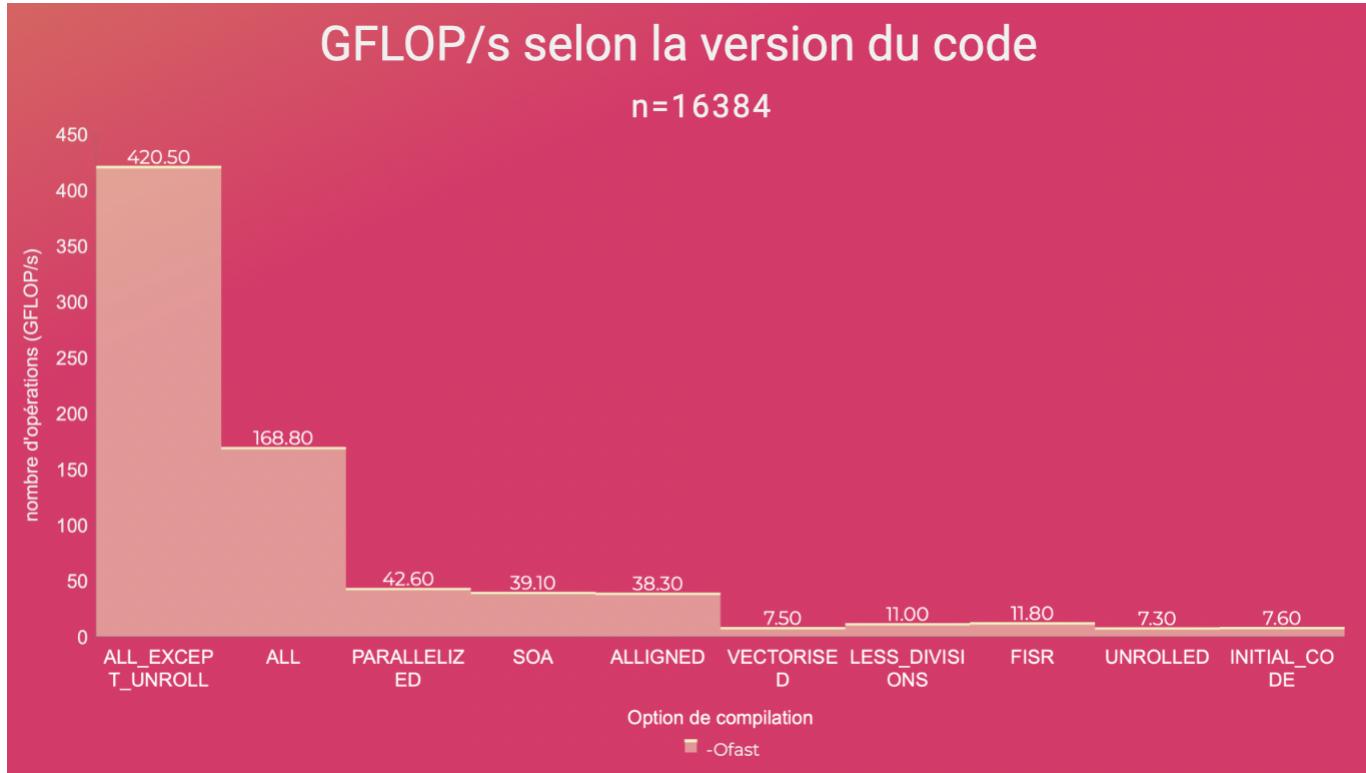
Toutes ces améliorations nous permettent de conclure que les optimisations plus agressives du niveau `-O2` sont globalement très efficaces.

Il est cependant important de noter que l'efficacité de ses optimisations dépend toujours du code, comme en témoigne la diminution de la vitesse d'exécution observée dans la version **PARALLELIZED**.



Il n'y a pas beaucoup eu de changements significatifs observés au niveau d'optimisation -O3.

Seulement une amélioration notable sur la version ALL\_EXCEPT\_UNROLL qui voit sa vitesse d'exécution augmenter d'environ 13 GFLOP/s.



Au niveau `-Ofast`, on remarque tout de suite que toutes les versions voient leur vitesse d'exécution augmenter.

J'attire votre attention sur les version SOA et ALLIGNED (aussi une SOA).

Cela car celles-ci n'avaient pas vu leur vitesse d'exécution augmenter par rapport au code initial jusqu'à présent. Ici, le code initial est à 7.6 GFLOP/s alors que SOA et ALLIGNED sont à environ 39 GFLOP/s.

Les versions VECTORISED et UNROLLED n'auront pas été des versions réussies de ce projet puisqu'elles ne montrent toujours pas d'amélioration par rapport au code initial.

Néanmoins, j'ai pris la décision de les laisser dans le code source et dans ce rapport, car j'avais initialement pensé qu'elles seraient efficaces.

De surcroît, je pense les avoir réalisé conformément aux techniques vues en cours.

# Bilan

En somme, il n'existe pas de méthode d'optimisation universelle qui fonctionnent avec tous les codes.

L'efficacité de ces méthodes dépend non seulement du code en question, mais également du compilateur utilisé et de l'architecture de la machine cible.

Ce projet m'a avant tout permis de réaliser, grâce aux mesures que j'ai faite, à quel point on peut faire varier la vitesse d'exécution d'un code en gardant le même résultat final.

Je garderai ces méthodes en tête tout au long de ma carrière.

Je remercie sincèrement M. Jalby, M. Ibnamar et M. Bollore pour la passion qu'ils transmettent et la qualité de leur enseignement.