



Université de Reims Champagne-Ardenne

**Master 2 : Calcul Haute Performance et Simulation**

**Année universitaire : 2023/2024**

---

# Optimisation de codes par des LLM

---

**Hafsa DEMNATI**

Encadré par **M. Luiz Angelo STEFFENEL**

**CHPS1004 - Projet**

# Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>  | <b>2</b>  |
| <b>2</b> | <b>Matériel et méthode</b>   | <b>2</b>  |
| 2.1      | Comparaison des LLMs Llama-2 et GPT-3 pour la génération de<br>kernels HPC . . . . . | 3         |
| 2.2      | Choix de modèles et d'un premier prompt . . . . .                                    | 3         |
| 2.3      | Codes choisis . . . . .  | 4         |
| 2.4      | ROMEO, supercalculateur pour les tests . . . . .                                     | 5         |
| <b>3</b> | <b>Résultats</b>   | <b>5</b>  |
| 3.1      | Exécution séquentielle . . . . .   | 6         |
| 3.2      | Génération des codes, erreurs rencontrées . . . . .                                  | 6         |
| 3.3      | Exécution parallèle avec les codes LLM . . . . .                                     | 7         |
| 3.4      | Comparaison des performances et speed-ups . . . . .                                  | 9         |
| <b>4</b> | <b>Discussion</b>  | <b>10</b> |
| 4.1      | Capacité du LLM à se corriger . . . . .  | 11        |
| 4.2      | Capacité du LLM à générer du code fonctionnel . . . . .                              | 11        |
| 4.3      | Capacité du LLM à garder le contexte . . . . .                                       | 11        |
| 4.4      | Comparaison entre les deux LLM . . . . .   | 11        |
|          | <b>Conclusion</b>  | <b>11</b> |
|          | <b>Bibliographie</b>   | <b>13</b> |
| <b>A</b> | <b>Erreurs à l'exécution MPI</b>   | <b>13</b> |
| <b>B</b> | <b>Erreurs de compilation Cuda</b>   | <b>13</b> |

# 1 Introduction

## Les LLM

Les grands modèles de langage (LLM) sont des outils d'intelligence artificielle capables de comprendre et de générer du texte en langage naturel. Grâce à des architectures avancées et à des volumes massifs de données d'entraînement, ces modèles peuvent effectuer diverses tâches telles que la traduction, la rédaction de contenu et la programmation. Les LLM représentent une avancée significative dans le domaine de l'interaction homme-machine, offrant des possibilités inédites en termes d'automatisation et d'optimisation des processus.

## Objectif du projet

L'objectif du projet présenté dans ce rapport est d'analyser la performance de codes parallélisés par une IA **LLM**. Avec l'émergence récente d'outils offrant une interaction homme-machine particulièrement intéressante, il est crucial de mesurer les performances en programmation pour évaluer les propositions possibles. Nous détaillerons les méthodes utilisées, les codes choisis, une comparaison de deux modèles LLM, ainsi que les résultats obtenus.

Pour ce faire, nous avons sélectionné plusieurs sujets d'un hackathon de programmation parallèle, proposant des codes sources séquentiels en C ou C++. Parmi ces problèmes, nous en avons choisi neuf pour tester des parallélisations **multicœur** (OpenMP), **multinœud** (MPI) et **GPU** (Cuda).

Après la sélection, nous avons demandé à deux LLM de paralléliser les codes afin d'identifier les erreurs les plus courantes et de tester leurs performances. Pour disposer d'un point de référence clair, nous avons également étudié un article traitant de la génération de kernels HPC, réalisant un travail de recherche similaire.

Les codes et scripts du projet se trouvent sur un dépôt git<sup>1</sup>.

## 2 Matériel et méthode

Dans cette première section du rapport, nous examinerons brièvement l'article cité en introduction. Ensuite, nous présenterons les LLM que nous avons retenus pour l'étude, en détaillant leurs architectures. Enfin, nous aborderons également le sujet des prompts choisis pour la génération des codes, ainsi que les codes retenus.

---

<sup>1</sup>[https://github.com/hafsa-dmmt/llm\\_optimization](https://github.com/hafsa-dmmt/llm_optimization)

## 2.1 Comparaison des LLMs Llama-2 et GPT-3 pour la génération de kernels HPC

Dans cette première sous-section, nous étudierons un article [4] portant sur la génération de kernels HPC à l'aide de **Llama-2**<sup>2</sup> et **GPT-3**<sup>3</sup>.

Dans cet article, les auteurs évaluent l'utilisation du modèle open-source **Llama-2** pour générer des noyaux HPC bien connus (tels que AXPY, GEMV et GEMM) sur différents modèles de programmation parallèle et langages (comme C++ avec OpenMP, OpenMPOffload, OpenACC, CUDA, HIP ; Fortran avec OpenMP, OpenMPOffload, OpenACC ; Python avec numpy, Numba, pyCUDA, cuPy ; et Julia avec Threads, CUDA.jl, AMDGPU.jl). Ils s'appuient sur leur travail précédent basé sur **OpenAI Codex**, qui est un descendant de **GPT-3**, pour générer des noyaux similaires avec des instructions simples via *GitHub Copilot*.

L'objectif est de comparer l'exactitude de **Llama-2** avec celle de leur modèle de base **GPT-3** en utilisant une métrique similaire. **Llama-2** présente un modèle simplifié qui montre une précision compétitive, voire supérieure. Les codes générés par **Copilot** sont plus fiables mais moins optimisés, tandis que ceux générés par **Llama-2** sont moins fiables mais plus optimisés lorsqu'ils sont corrects.

En ce qui concerne le choix des **prompts**, tout comme dans leurs précédentes recherches [2], ils ont été simples ; tous les prompts suivaient la structure suivante :

Create 3 code suggestions using the following parameters: <Programming Language> <Programming Model> <Kernel> <Keyword>.

Ce choix de prompt est crucial car il détermine comment le LLM générera le code demandé en se basant sur les informations fournies.

## 2.2 Choix de modèles et d'un premier prompt

Les modèles performants, cités dans la génération de codes et accessibles sont nombreux : Llama, GPT, Copilot, Mistral, Gemini,... Afin de ne pas nous étaler, nous avons décidé de faire nos tests en n'utilisant que Llama et GPT-3. Pour Llama, nous avons commencé à générer les codes à partir du lab Perplexity AI<sup>4</sup> et en choisissant le modèle **llama-3-70b-instruct**, qui a été développé par des équipes de Meta AI[3]. Pour GPT-3[1], nous avons utilisé la version publique disponible sur le site officiel<sup>5</sup>.

### Premier prompt

Comme nous l'avons vu précédemment, le choix du prompt, qui doit être précis et clair, est une tâche importante. Nous avons décidé de demander à nos LLM,

---

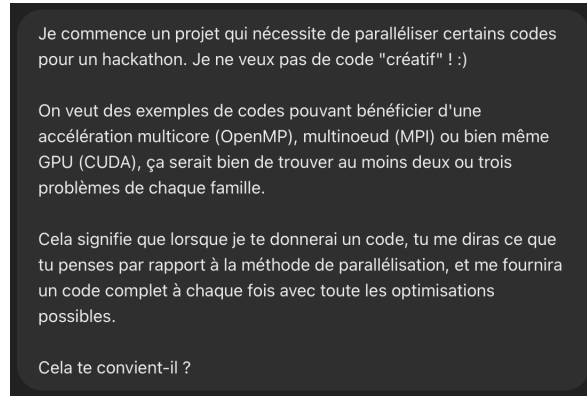
<sup>2</sup>Llama-2

<sup>3</sup>OpenAI

<sup>4</sup><https://labs.perplexity.ai>

<sup>5</sup>Chat OpenAI

en plus de la génération du code, de ne pas chercher à être créatifs et de ne pas réinventer de méthodes à partir des codes que nous avons choisis. Il est aussi important de fournir un contexte très précis pour que les résultats obtenus ne soient pas hors sujet ou trop libres. Nous voulions également que ce soit le modèle qui choisisse le modèle de parallélisation le plus adapté. Afin que les deux modèles fournissent des codes "équivalents", le choix de la technologie a été uniquement demandé à GPT-3.



Je commence un projet qui nécessite de paralléliser certains codes pour un hackathon. Je ne veux pas de code "créatif" ! :)

On veut des exemples de codes pouvant bénéficier d'une accélération multicore (OpenMP), multinoeud (MPI) ou bien même GPU (CUDA), ça serait bien de trouver au moins deux ou trois problèmes de chaque famille.

Cela signifie que lorsque je te donnerai un code, tu me diras ce que tu penses par rapport à la méthode de parallélisation, et me fournira un code complet à chaque fois avec toute les optimisations possibles.

Cela te convient-il ?

Figure 1: Premier prompt du projet dans la conversation avec GPT-3.

Après ce prompt très simple, nous avons seulement procédé de la même manière que dans l'article étudié, c'est à dire que nous avons envoyé le code à optimiser, et la technologie associée. Pour les codes MPI et Cuda nous avons aussi demandé à générer des makefile associés et des fichiers bash pour la soumission de job à l'aide de slurm.

## 2.3 Codes choisis

| Code                             | Technologie | Année | Nb caractères |
|----------------------------------|-------------|-------|---------------|
| ShellSort                        | OpenMP      | 2011  | 1112          |
| Dijkstra                         | OpenMP      | 2014  | 2547          |
| Jacobi Laplace for Heat Transfer | OpenMP      | 2021  | 3481          |
| Karatsuba                        | MPI         | 2015  | 4051          |
| Fibonacci numbers                | MPI         | 2009  | 888           |
| K-Means Clustering Problem       | MPI         | 2015  | 3476          |
| Finite Different Method          | MPI         | 2019  | 4545          |
| Brute Force Password Cracking    | Cuda        | 2019  | 1954          |
| Maximum Sum Subsequence          | Cuda        | 2019  | 830           |

Table 1: Codes choisis et technologie associée

Comme expliqué plus tôt, pour ce projet de recherche, nous avons choisi de

traiter des problèmes proposés par un hackathon<sup>6</sup>. Ce hackathon, proposé par la UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL, le LABORATÓRIO NACIONAL DE COMPUTAÇÃO CIENTÍFICA et la UNIVERSIDADE FEDERAL FLUMINENSE, est à destination d'étudiants en fin de licence, voire en première année de master. Ce sont des codes en théorie très étudiés et "facilement" parallélisables.

Le choix des codes à optimiser dépendait de plusieurs facteurs. Un résumé des choix est visible tableau 1.

- Le nombre de caractères : chaque LLM a un nombre de caractères maximal par prompt. En dehors de la limite imposée par l'interface nous permettant d'interagir avec le modèle, plus un prompt est long, plus le LLM a des risque de perdre le contexte de base et ne plus trouver le lien entre les tokens qu'il analyse. Afin d'avoir des résultats et des codes générés correctement, nous nous sommes limités à 4500 caractères au maximum, essayant de trouver une majorité de code bien en dessous de cette limite fixée.
- Des codes célèbres : chaque modèle étant entraîné sur des données existantes, et donc du code existant, nous avons choisi une cohorte de problèmes qui étaient bien connus (dans le monde académique notamment), étudiés, et optimisés depuis plusieurs années. Cela permettrait aux LLM de ne pas avoir à réinventer des optimisations et se baser sur des codes qu'il connaît potentiellement déjà.
- La possibilité de tester les codes avec différents paramètres, afin de tester les optimisations avec différentes charges de travail.

## 2.4 ROMEO, supercalculateur pour les tests

Pour nos tests, nous avons choisi d'utiliser les outils fournis par l'Université de Reims Champagne-Ardenne. En effet, ils offrent l'accès à plusieurs ressources, et notamment à un supercalculateur : le Centre de Calcul Régional ROMEO<sup>7</sup>.

Ce supercalculateur nous permet de lancer du code OpenMP, MPI et Cuda à l'aide des modules déjà installés sur les noeuds de calcul. Une installation slurm nous permet aussi de lancer des tests en théorie facilement.

## 3 Résultats

Dans cette section, nous présenterons les résultats obtenus en générant les codes suivant les méthodes détaillées. Nous analyserons les erreurs rencontrées pour déterminer si elles sont systémiques. Nous évaluerons également l'exécution des codes afin de vérifier s'ils sont exploitables et produisent des résultats satisfaisants.

---

<sup>6</sup>Hackathon édition 2010

<sup>7</sup><https://romeo.univ-reims.fr/>

### 3.1 Exécution séquentielle

Dans cette section, nous présenterons les résultats obtenus avec les codes séquentiels fournis par le marathon.

| Code                             | Params                          | Temps (s)  |
|----------------------------------|---------------------------------|------------|
| ShellSort                        | <i>shellsort.in</i> , 100000000 | 107.728864 |
| Dijkstra                         | 50 20 1000000                   | 0.000134   |
| Jacobi Laplace for Heat Transfer | 2048,15000                      | 198.854717 |
| Karatsuba                        | $0 \leq A, B \leq 10^{22}$      | 41.674410  |
| Fibonacci numbers                | $0 \leq N \leq 10^5$            | 3.304385   |
| K-Means Clustering Problem       | 65536 16 1024 10 65536          | 55.228630  |
| Finite Different Method          | 0.01 0.25 20.0 20.0 20.0        | 99.938582  |
| Brute Force Password Cracking    | 32-hexadecimal                  | 70.904656  |
| Maximum Sum Subsequence          | sum.in                          | 41.522873  |

Table 2: Temps (en secondes) de l'exécution des codes séquentiels

Il a été assez simple de lancer les codes ; la documentation était fournie et un Makefile permettait de compiler rapidement et efficacement les codes. Le seul point négatif était le manque d'informations sur les paramètres d'entrée pour certains programmes.

Quelques modifications ont tout de même été apportées :

- Intégration d'une mesure du temps d'exécution dans tous les codes.
- Passage des paramètres en ligne de commande. Certains codes attendaient une saisie sur l'entrée standard pour récupérer les paramètres, ce qui n'était pas pratique pour le lancement de jobs avec Slurm.

### 3.2 Génération des codes, erreurs rencontrées

Les erreurs rencontrées lors de la génération de code avec un LLM peuvent être divisées en deux catégories :

- **Erreurs syntaxiques** : ces erreurs peuvent, par exemple, être caractérisées par des oublis de caractères ou des caractères en trop.
- **Erreurs de programmation** : ces erreurs peuvent, par exemple, être caractérisées par des oublis de paramètres dans des fonctions, des rajouts de fonctions non existantes, des appels à des variables non existantes, etc.

Dans un premier temps, il est important de noter qu'aucune erreur syntaxique n'a été introduite. Cela a été surprenant car c'était une chose à laquelle nous nous attendions le plus.

Concernant l'autre type d'erreur, listées dans le tableau 3, on peut voir qu'il s'agit souvent des mêmes : des références à des variables non initialisées

| Erreur   | Code      | GPT-3 | Llama-2 |
|--|-----------|-------|---------|
| identifiant "true" is undefined <i>oubli bibliothèque</i>                  | Shellsort | x     | v       |
| 'for' loop initial declarations  | Shellsort | v     | v       |
| integer from pointer   | Shellsort | x     | v       |
| identifiant(s) undefined   | Dijkstra  | x     | v       |
| identifiant "true" is undefined <i>oubli bibliothèque</i>                  | Karastuba | v     | v       |
| identifiant "true" is undefined <i>oubli bibliothèque</i>                  | Karastuba | v     | v       |
| identifiant(s) undefined   | Karastuba | x     | v       |
| integer operation result is out of range                                   | Fibonacci | v     | v       |
| undefined behavior   | Fibonacci | x     | v       |
| integer overflow in expression   | Fibonacci | x     | v       |
| taking the address of a register variable is not allowed                   | MDF       | v     | v       |
| calling <code>_host_</code> function from a <code>_global_</code> function | Password  | v     | v       |
| identifiant " " is undefined in device code                                | Password  | v     | v       |

Table 3: Erreurs algorithmiques fréquentes lors de la génération de code pour le projet

ou inexistantes dans le code, des oublis de bibliothèques (notamment pour les booléens), des erreurs liées aux adressages mémoire. Ce sont des erreurs que les LLM ont ensuite corrigées d’eux-mêmes (dans la majorité des cas). Aucune erreur n’a été corrigée à la main, tout a systématiquement été demandé aux LLM.

En plus des erreurs de compilation, il y avait des erreurs à l’exécution. Ces dernières, plus verbeuses, sont visibles à l’annexe 4.4. On peut cependant noter qu’il y a eu énormément d’erreurs de segmentation, dues à des tableaux mal initialisés ou des types de variables inadaptés (par exemple, des *int* au lieu de *size\_t*).

De nombreuses erreurs ont été rencontrées en MPI, principalement à cause de communications entre nœuds mal gérées. De plus, lorsque les erreurs étaient réglées et que nous avions un speed-up impressionnant, les résultats calculés par les programmes n’étaient plus corrects. Le code généré n’est donc pas forcément correct même s’il s’exécute et termine sans erreur.

### 3.3 Exécution parallèle avec les codes LLM

Le test des codes a été effectué dans l’ordre des technologies : d’abord les codes OpenMP, puis MPI, et enfin CUDA. Nous verrons dans la suite que les exécutions CUDA n’ont pas pu aboutir à des résultats, et qu’en MPI, les résultats étaient très mitigés également.



## OpenMP

### Allocation des ressources :

```
1 salloc -N 1 --exclusive -p short
```

Pour OpenMP, il a été plus simple d’avoir un noeud en mode ”interactif” pour nos tests. Sur chaque noeud nous avons 24 threads à dispositions pour OpenMP. Les tests se sont alors déroulés pour 1, 2, 4, 8, 16 et 20 noeuds. En général, nous n’avons pas été jusque 24 car nous atteignons déjà un plateau autour de 20 threads, ou les résultats n’étaient plus intéressants.

## MPI

### Allocation des ressources :

```
1 #!/bin/bash
2
3 #SBATCH --job-name=code_mpi
4 #SBATCH --output=code_mpi_%j.out
5 #SBATCH --error=code_mpi_%j.err
6 #SBATCH --ntasks=4
7 #SBATCH --cpus-per-task=1
8 #SBATCH --mem-per-cpu=1024
9 #SBATCH -p short
10
11 source env.sh
12
13 mpirun -np 4 ./code_mpi [params]
```

Pour chaque job utilisant du MPI, nous avons comme base ce fichier *job.sh*. Nous avons alloué 4 noeuds pour tester les codes sur 1, 2 et 4 noeuds. Nous verrons dans la suite que ça n’a pas été concluant.

## Cuda

### Allocation des ressources :

```
1 #!/bin/bash
2
3 #SBATCH --job-name=code_cuda
4 #SBATCH --output=code_cuda_%j.out
5 #SBATCH --error=code_cuda_%j.err
6 #SBATCH --gres=gpu:1
7 #SBATCH --ntasks=1
8 #SBATCH --cpus-per-task=1
9 #SBATCH --mem-per-cpu=4G
10 #SBATCH -p short
11
12 source env.sh
13
14 ./code_cuda [params]
```

Nous avons à chaque fois alloué un GPU pour seulement tester le code.

### 3.4 Comparaison des performances et speed-ups

#### OpenMP

Dans un premier temps, on peut dire que les résultats OpenMP, du point de vue de la recherche dans ce projet, ont été satisfaisants. En effet, bien que dans certains cas les speed-ups n'aient pas été très bons, nous avons des codes fonctionnels qui donnaient pratiquement toujours le résultat attendu. Nous détaillerons ces points ci-dessous avec les courbes de temps associées à chaque code.

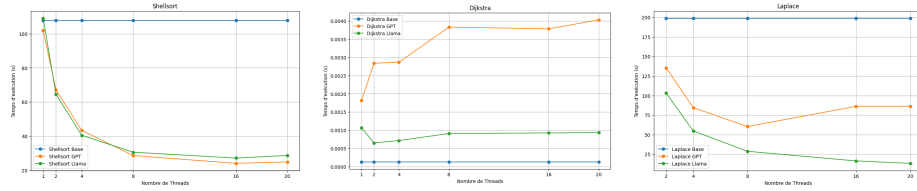


Figure 2: (a) Comparaison des temps OpenMP obtenue pour 1, 2, 4, 8, 16 et 20 threads avec les codes générés par CodeLlama et GPT-3 pour le problème **Shellsort**. Pour ce code, on peut voir des performances similaires entre GPT-3 et CodeLlama. (b) Comparaison des temps OpenMP obtenue pour 1, 2, 4, 8, 16 et 20 threads avec les codes générés par CodeLlama et GPT-3 pour le problème **Dijkstra**. Les résultats obtenus par GPT-3 sont **faux**, contrairement à ceux de CodeLlama. (c) Comparaison des temps OpenMP obtenue pour 1, 2, 4, 8, 16 et 20 threads avec les codes générés par CodeLlama et GPT-3 pour le problème **Laplace**. Les temps n'apparaissent pas pour **1 thread** car l'exécution, des deux côtés, s'est terminée par un **segfault**.

#### MPI

Comme on peut le voir dans le tableau 4, les résultats d'exécution MPI montrent plusieurs choses :

- "**Segfault**" ou "**No result**" : certains codes ont tourné à l'infini ou se sont terminés par des erreurs de segmentation. Cela est très probablement dû à une mauvaise gestion des communications.
- Speed up impressionnants à 1 nœud : certains codes bénéficient d'une accélération dès 1 nœud. Ce n'est pas normal, et en vérifiant, nous nous rendons compte que nous n'obtenons pas le bon résultat. Encore une fois, cela est probablement dû à une mauvaise gestion des communications ou à des erreurs algorithmiques qui font que les calculs ne sont pas effectués correctement.

Le seul code pour lequel nous avons obtenu des speedups exploitables pour les deux LLM est le code de MDF. Cependant, il n'est pas possible à première vue de vérifier que les résultats calculés sont corrects.

| Code               | LLM       | Nombre de noeuds | Temps (s) |
|--------------------|-----------|------------------|-----------|
| Karatsuba          | CodeLlama | 1                | 2.543449  |
| Karatsuba          | CodeLlama | 2                | 4.063329  |
| Karatsuba          | CodeLlama | 4                | 4.399016  |
| Karatsuba          | GPT-3     | 1, 2, 4          | No Result |
| Fibonacci          | CodeLlama | 4                | 0.051537  |
| Fibonacci          | GPT-3     | 4                | 1.937573  |
| K-Means Clustering | CodeLlama | 1, 2, 4          | Segfault  |
| K-Means Clustering | GPT-3     | 1, 2, 4          | Segfault  |
| MDF                | CodeLlama | 1                | 4.136894  |
| MDF                | CodeLlama | 2                | 2.693395  |
| MDF                | CodeLlama | 4                | 1.455700  |
| MDF                | GPT-3     | 1                | 3.899416  |
| MDF                | GPT-3     | 2                | 2.461489  |
| MDF                | GPT-3     | 4                | 2.279228  |

Table 4: Temps (en secondes) de l’exécution des codes MPI

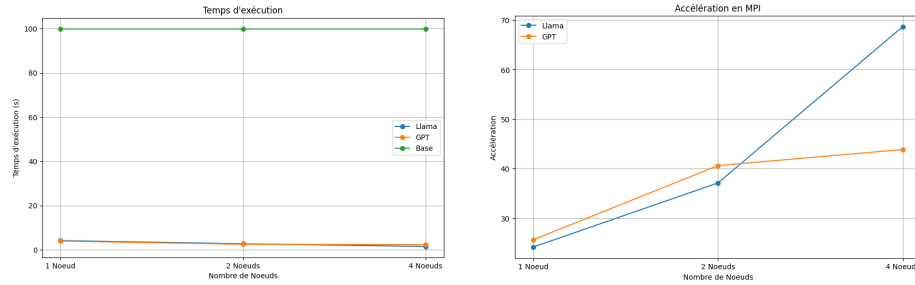


Figure 3: (a) Temps d’exécution en seconde pour 1, 2, 4 noeuds (CodeLlama et GPT-3). (b) Graphique montrant l’accélération fournie par MPI.

## Cuda

Comme on peut le voir dans le tableau 5, nous n’avons obtenu aucun résultat exploitable en CUDA. Peut-être que les LLM ont plus de facilité à optimiser un kernel déjà existant plutôt que de les écrire *from scratch*. Il faudrait passer plus de temps sur cette partie pour en déterminer les causes.

## 4 Discussion

Les erreurs rencontrées lors de la génération des codes avec les LLM sont variées. Les résultats montrent que les LLM peuvent générer des codes exploitables mais avec des erreurs fréquentes qu’il est nécessaire de corriger manuellement. Les performances des codes parallélisés seront comparées avec les versions séquentielles pour évaluer les gains obtenus.

| Code                    | LLM       | Temps (s)                   |
|-------------------------|-----------|-----------------------------|
| Password Cracking       | CodeLlama | 0.000000 <i>"Not Found"</i> |
| Password Cracking       | GPT-3     | Infinite Loop               |
| Maximum Sum Subsequence | CodeLlama | Infinite Loop               |
| Maximum Sum Subsequence | GPT-3     | No Functional Code          |

Table 5: Temps (en secondes) de l'exécution des codes Cuda

#### 4.1 Capacité du LLM à se corriger

Durant la génération de code, lorsque des erreurs étaient rencontrées, il a été demandé aux LLM de s'auto-corriger ; cela a mené plusieurs observations.

Ces observations, qui sont alors personnelles, montrent que CodeLlama a de meilleures capacités à se corriger que GPT-3. Cela peut être simplement expliqué par le fait que CodeLlama a été en grande partie entraîné sur du code. Ce LLM a aussi moins tendance à se perdre ou à tourner en rond dans le chat.

#### 4.2 Capacité du LLM à générer du code fonctionnel

Cette fois-ci, sur la génération d'un code fonctionnel, c'est plutôt GPT-3 qui l'a emporté. En effet, il a commis moins d'erreurs. Cependant, lorsqu'il en a commis, il n'arrivait pas à les détecter correctement et les corriger, contrairement à CodeLlama. Par exemple, dans l'un des codes, une erreur d'oubli de bibliothèque était présente ; GPT-3 insistait sur le fait de changer la déclaration d'une variable qui selon lui était responsable des erreurs à l'exécution. Il a fallu passer à un autre code car la correction n'évoluait plus et le LLM introduisait des erreurs supplémentaires dans le code.

#### 4.3 Capacité du LLM à garder le contexte

Enfin, il n'y a aucun doute sur le fait que CodeLlama est plus simple à utiliser du fait que ce LLM garde le contexte plus facilement que GPT-3.

Son inférence est beaucoup plus rapide et précise lorsque l'on a une requête particulière. Il est aussi capable de donner des réponses ouvertes. GPT-3 au contraire donne souvent des pistes très générales lors de ses générations de code, avec des conseils. Il faut être très insistant parfois pour obtenir des résultats convenables.

#### 4.4 Comparaison entre les deux LLM

Pour conclure sur cette étude comparative, il est certain qu'utiliser CodeLlama pour des problématiques de code est beaucoup plus intéressant. Cependant, il peut être aussi judicieux d'utiliser les deux pour faire des corrections de code croisées, ou bien même avoir plusieurs pistes de code.

## Conclusion

Les LLM comme GPT-3 et Llama-2 ont, comme on a pu le voir dans ce rapport, le potentiel d'optimiser les codes pour des tâches parallèles. Cependant, les erreurs générées montrent qu'un post-traitement manuel est souvent nécessaire pour assurer la fiabilité des codes. Des travaux futurs pourraient se concentrer sur l'amélioration des prompts et sur des modèles de vérification automatique des codes générés. Il peut être intéressant de s'en servir d'aide, mais pas forcément comme un outil entier pour la génération de code.

On a aussi pu voir dans le rapport qu'il était beaucoup plus simple de mettre en place des optimisations OpenMP que MPI ou Cuda. Cela peut être aussi expliqué par le fait qu'OpenMP nécessite moins de ressources hardware et que les directives sont très simples à ajouter au code (notamment en détectant les boucles, etc.).

Ainsi, les résultats ont aussi grandement été influencés par l'installation (aussi bien matérielle que logicielle) de ROMEO. Ayant rencontré beaucoup de difficultés sur des questions de connexion aux serveurs, d'allocation de noeuds, etc., nous aurions sûrement pu aller plus loin avec une installation plus stable.

Sur une note plus personnelle, ce projet a mis en avant une difficulté qui s'éloigne grandement des questions de compréhension de code ou IA. En effet, gérer 27 codes différents et les multiples compilations/exécutions, après avoir épluché les sujets sur plusieurs années de marathons différents n'a pas été une tâche simple. Cependant, c'était un sujet très intéressant qui peut encore être creusé.

## Pistes pour la suite

- Optimisation de codes plus complexes/avec d'autres technologies.
- Comparer les résultats obtenus avec les temps des gagnants des hackathons.
- Faire de la correction croisée entre CodeLlama et GPT-3 pour débloquer les deux LLM.
- Inverser l'ordre du début de projet et demander à CodeLlama de choisir les technologies pour chaque code au lieu de le faire avec GPT-3.

## Bibliographie

- [1] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.
- [2] William Godoy et al. “Evaluation of openai codex for hpc parallel programming models kernel generation”. In: *Proceedings of the 52nd International Conference on Parallel Processing Workshops*. 2023, pp. 136–144.
- [3] Hugo Touvron et al. “Llama: Open and efficient foundation language models”. In: *arXiv preprint arXiv:2302.13971* (2023).
- [4] Pedro Valero-Lara et al. “Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation”. In: *arXiv preprint arXiv:2309.07103* (2023).

## A Erreurs à l’exécution MPI

```
-> Segmentation fault

-> Abort(201960966) on node 2 (rank 2 in comm 0): Fatal error in
PMPI_Isend: Invalid rank, error stack:
PMPI_Isend(157): MPI_Isend(buf=0x7fc6c4b0b010, count=1048573, MPI_INT,
dest=4, tag=0, MPI_COMM_WORLD, request=0x7ffd45047840) failed
PMPI_Isend(94): Invalid rank has
value 4 but must be nonnegative and less than 4
Abort(201960966) on node 3 (rank 3 in comm 0):
Fatal error in
PMPI_Isend: Invalid rank, error stack:
PMPI_Isend(157): MPI_Isend(buf=0x7f645e32c010, count=1048573,
MPI_INT, dest=5, tag=0, MPI_COMM_WORLD, request=0x7ffcbeba2480) failed
PMPI_Isend(94): Invalid rank has
value 5 but must be nonnegative and less than 4

-> Abort(738806533) on node 2 (rank 2 in comm 0): Fatal
error in PMPI_Comm_size: Invalid communicator, error stack:
PMPI_Comm_size(110): MPI_Comm_size(comm=0x0, size=0x602050) failed
PMPI_Comm_size(67): Invalid communicator
```

## B Erreurs de compilation Cuda

```
nvcc -O3 -lssl -lcrypto password_bf.cu -o password_bf
password_bf.cu(27): warning: address of a host variable "letters"
cannot be directly taken in a device function
```

```
password_bf.cu(28): warning: a host variable "letters"
cannot be directly read in a device function

password_bf.cu(61): warning: variable "hash2"
was declared but never referenced

password_bf.cu(17): error: calling a __host__ function("MD5_Init")
from a __device__ function("md5") is not allowed

password_bf.cu(17): error: identifier "MD5_Init" is undefined in device code

password_bf.cu(18): error: calling a __host__ function("MD5_Update")
from a __device__ function("md5") is not allowed

password_bf.cu(18): error: identifier "MD5_Update" is undefined in device code

password_bf.cu(19): error: calling a __host__ function("MD5_Final")
from a __device__ function("md5") is not allowed

password_bf.cu(19): error: identifier "MD5_Final" is undefined in device code

password_bf.cu(27): error: calling a __host__ function("strlen")
from a __global__ function("iterateKernel") is not allowed

password_bf.cu(27): error: identifier "strlen" is undefined in device code

password_bf.cu(27): error: identifier "letters" is undefined in device code

password_bf.cu(28): error: identifier "letters" is undefined in device code

password_bf.cu(31): error: calling a __host__ function("memcmp")
from a __global__ function("iterateKernel") is not allowed

password_bf.cu(31): error: identifier "memcmp" is undefined in device code

12 errors detected in the compilation of "password_bf.cu".
make: *** [password_bf] Error 1
```