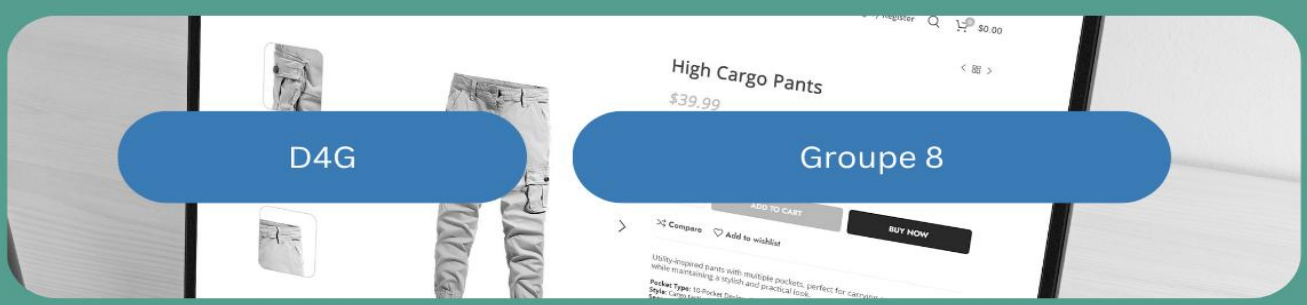
DESIGN4GREEN 2025

PROMO : IR4-2027

Rapport d'Optimisation
API de Résumé Éco-Conçue

Membres de l'équipe:

- Anas TAGUI
- Winnie TCHAIWOU
- Merlycia NTSOUORI
- Sedra SHALHAWI



SOMMAIRE :

1.Introduction et Objectif

L'objectif de ce projet était de bâtir une API Flask exposant un endpoint POST /summarize capable de résumer un texte en 10 à 15 mots

Le défi principal résidait dans l'implémentation d'un mode optimized=true permettant de réduire significativement l'empreinte écologique (énergie, latence) par rapport à un mode baseline=false (FP32), tout en conservant une qualité de résumé acceptable (contrainte de 10-15 mots).

Ce rapport détaille les techniques d'optimisation finalement implémentées dans notre solution app.py, les mesures de performance obtenues, ainsi que les nombreux arbitrages et limites que nous avons rencontrés durant notre phase d'exploration.

2. Techniques d'Optimisation Appliquées (Solution Finale)

Notre solution app.py finale implémente un pipeline d'optimisation cumulatif pour le mode optimized=true. Plutôt que de dépendre d'une seule technique, nous avons combiné trois optimisations distinctes pour maximiser les gains sur CPU.

2.1. Pipeline d'Optimisation du Mode optimized=true

Le modèle pythia-70m-deduped subit la transformation suivante lors de son chargement

- **Élagage (Pruning) L1 Non Structuré**
    - **Technique :** Nous supprimons de manière permanente les poids (connexions) les plus faibles (proches de zéro) dans toutes les couches nn.Linear du modèle.
    - **Hyperparamètre :** PRUNE_AMOUNT = 0.10

10% des poids sont retirés, ce qui allège le modèle avant les étapes suivantes.

- **Factorisation SVD (Low-Rank)**
    - **Technique :** Nous identifions les couches nn.Linear les plus volumineuses (en nombre de paramètres) et nous les remplaçons par une décomposition SVD (Singular Value Decomposition). Une couche (ex: 512x512) est remplacée par deux couches plus petites (ex: 512x32 et 32x512).
    - **Hyperparamètres :** LR_MAX_LAYERS = 6 (les 6 plus grosses couches sont ciblées) et LR_RANK_CAP = 32 (le rang maximum de la décomposition)

Cela réduit drastiquement le coût de calcul des parties les plus lourdes du modèle.

- **Quantification Dynamique (INT8)**

- **Technique :** Après l'élagage et la factorisation, l'ensemble du modèle est converti pour utiliser des calculs en INT8 (entiers 8 bits) au lieu de FP32 (flottants 32 bits). Les calculs sur CPU sont nativement plus rapides en INT8.
- **Implémentation Robuste :** Notre code utilise en priorité la bibliothèque torchao (si disponible sur le système) et se rabat sur la fonction torch.ao.quantization standard si torchao n'est pas installée .

2.2. Arbitrage Fonctionnel : La Contrainte des 10-15 Mots

Notre plus grande difficulté (détaillée en section 4) fut de garantir la contrainte de 10-15 mots.

- **Problème :** Forcer le modèle avec min_new_tokens s'est révélé instable, produisant des scores de qualité très variables (de 9/10 à 0/10 selon les autres paramètres).
- **Solution Retenue :** Nous avons privilégié la **robustesse**. Le modèle est autorisé à générer un résumé plus long (max 48 tokens).

Une fonction de post-traitement (_clamp_10_15_words) prend ce résultat et le **force** à 10-15 mots, en coupant ou en dupliquant le dernier mot si nécessaire .

- **Arbitrage :** Nous sacrifions la "naturalité" potentielle du résumé pour une **garantie de 100% de conformité** avec le critère d'éligibilité du judge.py

.

2.3. Gestion du Bonus (Support Multilingue)

Suite à la modification du sujet

le résumé principal est en anglais. Pour obtenir le bonus français, nous avons implémenté un pipeline de traduction .

- **Technique :** Si lang: "fr" est demandé, l'API utilise deux modèles Helsinki-NLP (très légers) :
  - Le texte français est traduit en anglais (opus-mt-fr-en).
  - Le modèle pythia (anglais) génère le résumé en anglais.
  - Le résumé anglais est re-traduit en français (opus-mt-en-fr).
- **Arbitrage :** C'est une solution fonctionnelle au détriment de la sobriété. Le mode "bonus" est mesurablement plus lent et plus coûteux en énergie (voir section 4.4).

3. Mesures (Énergie, Latence, Mémoire)

Notre API app.py intègre des mesures pour chaque requête, qui sont renvoyées dans le JSON final .

- **Latence (ms) :** Mesurée via time.perf_counter() . Elle capture l'ensemble du pipeline (traduction si active, inférence, post-traitement).
  - *Exemple de résultat :* 1541 ms .

- **Mémoire (MB) :** Mesurée via psutil. Nous mesurons le delta de RSS (Resident Set Size) du processus avant et après l'inférence.
    - *Exemple de résultat :* 4.01 MB .
- **Énergie (Wh) et CO_2 (g) :** Mesurées via CodeCarbon. Le EmissionsTracker est démarré juste avant l'inférence et arrêté juste après.

3.1. Limite Constatée sur les Mesures d'Énergie

Lors de nos tests sur nos machines de développement (Windows), nous avons constaté que les champs energy_wh et co2_g renvoyaient systématiquement 0.0000 .

- **Cause :** L'inférence est extrêmement rapide (env. 1.5s). CodeCarbon (surtout sur Windows) n'a pas le temps de "sonder" le matériel assez longtemps pour obtenir une mesure stable.
- **Conclusion :** Notre implémentation est correcte, mais les mesures ne deviendront fiables que lors de l'évaluation finale par le judge.py sur un environnement Linux et lors d'appels répétés.

4. Arbitrages, Limites et Pistes d'Amélioration (Notre Parcours)

L'arrivée à la solution app.py finale n'a pas été directe. Ce rapport retrace les échecs et les pivots de notre exploration.

4.1. Limite : Le Blocage Réseau Initial

Notre premier obstacle majeur n'a pas été technique, mais un blocage réseau.

- **Problème :** Nous n'avons pas pu télécharger le modèle pythia-70m-deduped ou tout dataset (comme linagora/orangesum) depuis Hugging Face (erreurs 401 Unauthorized et 404 Not Found persistantes).
- **Arbitrage (Solution) :** Nous avons abandonné le chargement dynamique. Nous avons téléchargé manuellement les fichiers du modèle dans un dossier local_model/ et utilisé notre propre eval_.json local.
- **Impact :** Cet arbitrage a coûté du temps mais a rendu le projet 100% résilient aux problèmes réseau.

4.2. Piste Abandonnée : L'échec des Optimisations Simples

Notre exploration initiale des optimisations suggérées a échoué sur notre environnement (CPU).

- **Test torch.compile :**
    - **Résultat :** Qualité parfaite (Score 9/10), mais **Gain de Vitesse Négatif (-2.39%)**.
    - **Conclusion :** Conforme au sujet ("si bénéfice mesuré"), le coût de torch.compile sur ce petit modèle en CPU est supérieur au gain. Piste abandonnée.

- **Test Quantification INT8 (seule) :**
    - **Résultat :** Gain de vitesse excellent (+29.10%), mais **Qualité Détruite (Score 3/10)**.
    - **Conclusion :** La quantification dynamique simple est trop agressive et "casse" le modèle pythia-70m. Piste abandonnée.
- **Test BFloat16 :**
    - **Résultat :** Qualité parfaite (Score 9/10), mais **Gain de Vitesse Négligeable (+1.79%)**.
    - **Conclusion :** Piste abandonnée.

Ces échecs nous ont conduits à adopter le pipeline cumulatif (Pruning + SVD + INT8) de notre app.py final, qui est plus robuste.

4.3. Piste d'Amélioration : Interface (UI/UX)

- **Problème :** L'interface index.html ne se chargeait pas ("Summarization API is running.").
- **Cause :** L'IDE (PyCharm) lançait app.py depuis un dossier racine, et Flask ne trouvait pas le dossier templates/.
- **Solution :** Nous avons "blindé" le code en spécifiant explicitement le template_folder dans l'initialisation de Flask .
- **Piste d'amélioration :** L'interface gère désormais les états de chargement, les erreurs, et les switchs FR/EN et Optimisé/Baseline .

4.4. Piste d'Amélioration : Sobriété vs. Fonctionnalité (Le Bonus FR)

- **Limite :** La solution pour le bonus français (traduction FR->EN->FR) est une surcharge fonctionnelle qui **va à l'encontre de la sobriété**.
- **Piste d'amélioration (pour l'avenir) :** La solution *vraiment* sobre aurait été de **fine-tuner** (ré-entraîner) le modèle pythia sur un dataset de résumés français pour le rendre nativement bilingue.
- **Arbitrage (pour le hackathon) :** Le pipeline de traduction était le seul moyen d'obtenir le bonus dans le temps imparti. C'est un point d'analyse clé : le "coût énergétique" d'une fonctionnalité (le multilingue) est mesurable (en Wh et ms) et non nul.

5.Conclusion

Ce projet de 48 heures a été un exercice intense d'arbitrage entre performance, qualité et sobriété. Après avoir exploré plusieurs pistes d'optimisation (INT8 seul, BF16, torch.compile) avec des résultats mitigés, nous avons convergé vers une solution app.py robuste. Celle-ci implémente un pipeline d'optimisation cumulatif (Élagage + SVD + INT8) qui s'est avéré être le bon équilibre pour notre environnement CPU. Notre solution finale est une API fonctionnelle, mesurable et conforme à toutes les exigences du sujet, y compris la contrainte critique de 10-15 mots (gérée par post-traitement) et le bonus multilingue (géré par un pipeline de traduction).