Calcul haute performance en Python pour la Science des Données: une vue d'ensemble

Pierrick Bruneau*, Oscar Castro**, Jean-Sébastien Sottet*

*LIST, 5 Avenue des Hauts-Fourneaux, L-4362 Esch-sur-Alzette
prenom.nom@list.lu,
https://www.list.lu
**Université du Luxembourg, 2 Avenue de l'Universite, L-4365 Esch-sur-Alzette
prenom.nom@uni.lu
https://www.uni.lu

Résumé. Python est devenu le langage préférentiel dans les domaines de la Science des Données et de l'Apprentissage Automatique. Cependant, les *data scientists* ne sont pas nécessairement des programmeurs expérimentés. Bien que Python leur permette d'implémenter rapidement leurs algorithmes, pour passer à l'échelle, l'efficacité du calcul devient un souci inévitable. Ainsi, tirer le meilleur parti des capacités de processeurs multi-coeur et de *Graphical Processing Units* (GPU) n'est généralement pas trivial. Dans cet article, nous présentons les principaux résultats d'un récent article de synthèse, conçu comme un document de référence permettant aux praticiens en Science des Données d'apprivoiser la richesse des outils et des techniques disponibles pour le langage Python. Nous mettons un accent particulier sur la détermination des principaux traits et caractéristiques distinctives des contributions dans ce domaine. Ce document peut aider les praticiens de la Science des Données dans leur choix d'outils, et les développeurs d'outils dans l'identification de manques potentiels dans les travaux existants.

1 Introduction

Python joue un rôle fondamental dans la Science des Données et l'Apprentissage Automatique, avec des bibliothèques de grande importance pratique telles que NumPy, Pandas, TensorFlow, et Scikit-learn. Cependant, l'interpréteur par défaut de Python, CPython, est réputé comme relativement lent, incitant au développement de bibliothèques haute performance dans des langages à typage statique tels que C++, Fortran, et CUDA. Les inefficacités inhérentes à CPython sont principalement dues à son système de typage d'objets dynamique et au surcoût d'exécution lors de l'invocation de fonctions de bibliothèque compilées (Ismail et Suh, 2018). Python, particulièrement son implémentation par défaut, CPython, présente des limites dues à son *Global Interpreter Lock* (GIL), impactant son efficacité en termes de *multi-threading* et d'accès concurrent, conduisant à une faible capacité de parallélisation des instructions. En réponse à ces limitations, de nombreux outils et stratégies pour la programmation haute performance en Python ont été proposés. L'objectif de ce document est de fournir une vue organisée

de ces derniers basée sur des scénarios d'utilisation communs dans le domaine de la Science des Données. Le présent article expose de manière résumée une récente revue narrative (Castro et al., 2023). Ce travail nous a permis d'identifier quelques grandes catégories d'outils et de techniques, qui s'alignent assez précisément avec nos scénarios d'utilisation. Nous avons choisi ces derniers afin de refléter les points de départ variés rencontrés par les *Data Scientists*. Ainsi, ces praticiens peuvent souhaiter mettre à l'échelle une implémentation en Python standard existante. Cette implémentation peut aussi fortement dépendre de bibliothèques numériques populaires, et nous montrons que cela implique des approches assez différentes. Enfin, le projet de développement peut encore se trouver sur le papier, le ou la *Data Scientist* souhaitant alors une efficacité computationnelle maximale.

Les sections de notre document reflètent ces scénarios, résumées par une table de recommandations. Dans cette communication, nous mettons en avant l'exposition de nos résultats. Pour les détails méthodologiques (e.g., sources d'information, mots-clés, critères d'exclusion), nous renvoyons le lecteur vers la revue narrative *in extenso* (Castro et al., 2023). Par ailleurs, nous traitons le cas des interpréteurs Python alternatifs à CPython dans une section dédiée.

2 Accélération de code Python standard

Dans ce scénario, le ou la *data scientist* a implémenté son algorithme en Python standard, c'est-à-dire en ne recourant peu ou pas à des bibliothèques tierces. Cette approche permet une bonne lisibilité du code, mais comme discuté en introduction, cela conduit à un logiciel souvent très lent. Les outils s'offrant alors aux praticiens peuvent être essentiellement regroupés en deux grandes catégories.

2.1 Parallélisation et distribution des tâches

Les tâches de Science des Données comprennent souvent des opérations itératives sur des blocs de données indépendants, comme le traitement d'images ou d'enregistrements de bases de données. Une manière bien établie d'utiliser la parallélisation comme moyen d'accélérer le code consiste à mettre en oeuvre le principe de *Single Instruction, Multiple Threads* (SIMT) grâce à des bibliothèques comme *Message Passing Interface* (MPI) et *Open Multi-Processing* (OpenMP), qui en Python sont accessibles via des *wrappers* tels que MPI4Py (Dalcín et al., 2005) and Pymp (Lassner, 2015). MPI fonctionne avec un modèle de mémoire distribuée et a été conçu à l'origine pour des langages comme C, C++, et Fortran, permettant aux ordinateurs exécutant des programmes parallèles d'échanger des messages sur un réseau distribué de machines. À l'inverse, OpenMP est adapté pour les processeurs multi-coeur et fournit des directives de compilateur qui sont moins intrusives que MPI, permettant la parallélisation au niveau des boucles, potentiellement imbriquées. De manière alternative, certaines bibliothèques en Python comme PyCOMPSs (Tejedor et al., 2017) agissent en tant que *wrappers* d'outils de parallélisation basés sur des tâches en C++, utilisant le plus souvent des décorateurs pour masquer la complexité d'utilisation de ces derniers. Les décorateurs sont un type d'instruction placé avant une définition de fonction, permettant de décrire sa parallélisation. Jug (Coelho, 2017) définit également des tâches au travers de fonctions, en utilisant le décorateur @task-generator pour analyser les graphes de dépendance de tâches et réaliser la parallélisation en exécutant plusieurs processus.

2.2 Compilation et transpilation

La transformation de programmes et la compilation *Just-In-Time* (JIT) est également une manière commune pour améliorer la performance d'exécution du code. Cython (Ewing et al., 2022) et Numba (Lam et al., 2015) sont respectivement les deux outils les plus représentatifs de ces deux sous-catégories. Cython agit à la fois comme une extension de langage et un compilateur, traduisant le code Cython de haut niveau en code C efficace et fournissant des mécanismes de parallélisation, tandis que Numba, un compilateur JIT, traduit le code Python en une représentation intermédiaire, qui est ensuite compilée par LLVM, générant ainsi du langage machine. Nuitka (Nuitka, 2022) traduit les instructions CPython en C++, s'intégrant avec l'interpréteur Python pour les parties qui ne peuvent pas être compilées, garantissant ainsi la compatibilité avec d'autres bibliothèques. Il revendique une automatisation complète, et génère des fichiers exécutables et du code pouvant être invoqué directement ou comme bibliothèques tierces. Pythran (Guelton et al., 2015) convertit le code Python en code C++ optimisé. Il effectue une analyse de code et des transformations sur l'arbre syntaxique induit par le code Python traité, afin de générer une version plus rapide, assurant la compatibilité avec les expressions NumPy et des optimisations comme la vectorisation de boucle appliquées via OpenMP.

3 Accélération en présence de bibliothèques tierces

Les bibliothèques numériques NumPy (Harris et al., 2020), Pandas (Pandas Development Team, 2022), et Scikit-learn (Pedregosa et al., 2011) sont les plus populaires pour faciliter les développements dans le domaine de la Science des Données et de l'Apprentissage Automatique (Castro et al., 2023). Pour accélérer un programme utilisant ces bibliothèques, des équivalents *drop-in*, qui peuvent exploiter plusieurs coeurs de processeur ou de GPU, ont été développés. Ils ont généralement le souci de se calquer sur l'API de la bibliothèque qu'ils remplacent, ne nécessitant donc que des modifications mineures du code existant.

3.1 NumPy

Le format de tableau numérique de NumPy sert de base à de nombreux projets en Science des Données, il est donc naturel que les tentatives d'accélérer l'exécution de NumPy aient également été nombreuses. Des bibliothèques comme Bohrium (Kristensen et al., 2013), CuPy (Ryosuke et al., 2017) et JAX (Bradbury et al., 2018) proposent des exécutions parallèles, des optimisations, et des solutions de calcul distribué comme alternatives à NumPy, avec des variations dans les approches implémentées et dans le degré de couverture de l'API de NumPy. Bohrium met l'accent sur la logique de distribution et le déploiement des opérations de NumPy sur différentes plateformes matérielles, avec une évaluation paresseuse et un regroupement d'opérations de tableau lorsque nécessaire dans Bohrium. CuPy exploite le langage Nvidia CUDA pour l'accélération GPU, avec un accent sur la gestion des échanges entre la mémoire principale et la mémoire GPU. JAX propose des transformations composables de programmes Python basées sur la syntaxe de NumPy, permettant notamment de vectoriser certaines opérations.

3.2 Pandas

Le format de *dataframe* Pandas permet de décrire facilement des objets selon des attributs de type hétérogènes ainsi que des séries temporelles, en faisant un standard *de facto* dans la pratique de la Science des Données. L'amélioration de son efficacité a ainsi été significativement étudiée dans la littérature. Modin (Petersohn et al., 2020) optimise les opérations sur les *dataframes* Pandas dans un environnement local ou de cluster. cuDF (Nvidia, 2022) permet d'exploiter les GPU en utilisant une API proche de Pandas. Alternativement, Polars (Polars, 2022) n'est pas explicitement une bibliothèque *drop-in* de Pandas, mais elle présente des similitudes significatives et nécessite des transformations mineures pour être utilisée aux mêmes fins.

3.3 Scikit-learn

SciPy (Virtanen et al., 2020) vise une large couverture des concepts mathématiques et statistiques en réutilisant le format de tableau de NumPy. Scikit-learn (Pedregosa et al., 2011) est construit sur la base de SciPy, avec pour but d'implémenter des modèles représentatifs de la littérature en apprentissage automatique. Cette bibliothèque essaie autant que possible de présenter une interface unifiée aux différents modèles proposés (e.g., *train* puis *fit*). cuML, qui fait partie de l'ensemble de bibliothèques RAPIDS (Nvidia, 2022), est basée sur le langage CUDA, exploitant les capacités du GPU pour accélérer les *pipelines* de Science des Données. Elle a pour but explicite de couvrir l'API de Scikit-learn de manière aussi transparente que possible, exploitant des données gérées par CuPy et les formats de *dataframe* cuDF. Les gains de performance sont significatifs selon des benchmarks du domaine. Dislib (Álvarez Cid-Fuentes et al., 2019) propose une API basée sur la notion d'estimateur similaire à Scikit-learn. Elle tire parti de PyCOMPSs pour traiter des données distribuées sur des clusters de calcul haute performance.

4 Frameworks structurants

Dans cette section, nous avons regroupé les outils les plus contraignants sur la structure d'un logiciel. A ce titre, ils doivent préférablement être utilisés dès le début d'un projet de développement. La complexité de ces outils, tant en termes d'architecture que d'utilisation avancée, fait qu'on parle souvent de *frameworks* à leur sujet. Nous avons identifié deux familles d'outils dans cette section : les *frameworks* d'apprentissage profond, et de calcul distribué basé sur des tâches.

4.1 Apprentissage profond

De nombreux modèles utiles dans la pratique de la Science des Données sont formalisés comme des *Directed Acyclic Graphs* (DAG), dont l'optimisation est typiquement parallélisables. Les deux frameworks présentés dans cette section offrent un support spécialisé pour de tels modèles, compilent les DAG et fonctions associés, et les chargent sur le GPU, permettant en principe une accélération significative. En revanche, ils imposent des paradigmes de programmation spécifiques, souvent divergents des constructions Python standard, et nécessitent

Technique	**Diff.**	**Gain**	**Outils**
Parallélisation et distribution des tâches	+/-	-	MPI4Py, Pymp , PyCOMPSs, Jug, *Pygion*, *PyKokkos*
Compilation et transpilation	-	-	Cython, Numba, Nuitka, *Pythran*, *Pyjion*, *Pyston-lite*, *Shed Skin*, *Transpyle*, *Pydron*, *Autoparallel*, *Hope*
Libraries *drop-in*	+/-	+	Bohrium, CuPy, JAX, *PyViennaCL*, *Distarray*, *D2O*, *DelayRepay*, *NumExpr*, *PyPacho*, Modin, CuDF, Polars, *Datatable*, *Vaex*, cuML, Dislib, *MLlib*
Apprentissage profond	++	++	Tensorflow, PyTorch, *Keras*, *Theano*, *MXNet*
Calcul distribué basé sur des tâches	++	++	Horovod, Ray, Dask, *torcpy*, *CharmPy*, *SCOOP*, *Parallel Python*, *Celery*, *Playdoh*, *Dace*, *Ipyparallel*, *Parsl*, *Tuplex*

TAB. 1 – *Résumé des approches d'optimisation de performances Python. La difficulté de mise en oeuvre et le gain de performance attendu moyens de chaque catégorie sont évalués de manière qualitative de - (moindre) à ++ (élevé). Les références en italique ne sont pas explicitement mentionnées dans ce document.*

un temps significatif pour être maîtrisés au-delà d'un usage basique. Tensorflow (Abadi et al., 2016) est un outil réputé pour l'apprentissage profond, offrant une API complète pour programmer des composants personnalisés. Torch, développé par Meta, est un framework d'apprentissage profond basée sur Lua. PyTorch (Paszke et al., 2019) est son portage en Python préservant les principes de Torch. PyTorch, avec 8M de téléchargements mensuels, rivalise en popularité avec TensorFlow, marquée par 15M de téléchargements mensuels. PyTorch excelle par sa bonne documentation et son API de haut niveau, offrant plus de contrôle sur l'exécution des processus d'apprentissage et facilitant la mise en oeuvre de boucles d'entraînement sophistiquées. Cependant, elle nécessite plus de code et présente des temps d'exécution plus longs par rapport à TensorFlow. TensorFlow et PyTorch ont des différences inhérentes dans la gestion des graphes computationnels, impactant leur capacité à gérer des données d'entrée de dimensionalité variable. Les graphes computationnels statiques de TensorFlow rendent difficile la gestion de cette situation, tandis que les graphes dynamiques de PyTorch permettent plus de flexibilité. Le débogage dans PyTorch est plus direct grâce à la définition dynamique des graphes. Enfin, notons que Tensorflow et PyTorch disposent de bibliothèques annexes (Probability et Pyro, respectivement) permettant d'accélérer l'inférence de modèles Bayésiens complexes.

4.2 Calcul distribué basé sur des tâches

En guise de transition vers le traitement des bibliothèques de calcul distribué, nous pouvons souligner Horovod (Sergeev et Balso, 2018), qui facilite l'utilisation de ressources distribuées aux utilisateurs de frameworks tels que Tensorflow, PyTorch ou MXNet. Cette bibliothèque permet de mettre à l'échelle de manière efficace des tâches d'apprentissage, notamment en

Nombre d'étoiles	**Outils**
De 0 à 100	PyCOMPSs, *Pygion*, *Pyjion*, *Pyston-lite*, *PyPacho*, *MLlib*, *Parallel Python*, *D2O*, *Distarray*, *Pydron*, DistNumPy, *Autoparallel*, *TorcPy*, *DelayRepay*, PyOMP, dislib, *PyKokkos*, *Playdoh*, PyPar, *PyViennaCL*
De 101 à 5000	*Transpyle*, Bohrium, cuML, Pymp, *CharmPy*, *Parsl*, *Dace*, *Hope*, Jug, *SCOOP*, MPI4Py, *Shed skin*, *Tuplex*, *datatable*, *Pythran*, *NumExpr*, *Ipyparallel*
Plus de 5000	cuDF, CuPy, Cython, *Vaex*, Numba, Modin, Nuitka, *Theano*, Dask, Horovod, polars, *MXNet*, *Celery*, JAX, Ray, *Keras*, PyTorch, TensorFlow

TAB. 2 – *Répartition des outils selon le nombre d'étoiles GitHub associé. Les terciles ont été choisis de manière à avoir 3 catégories équilibrées. Les références en italique ne sont pas explicitement mentionnées dans ce document.*

comparaison de l'API spécialisée fournie avec Tensorflow. Ray (Moritz et al., 2018) offre à la fois des primitives de bas niveaux pour l'exécution basées sur des tâches, et une API de haut niveau pour l'apprentissage profond et l'analyse de données. Il bénéficie d'une communauté active, et sert de base à certains outils mentionnés dans d'autres sections de ce document, par exemple Modin en Section 3.2. Dask (Dask Development Team, 2016) est étroitement intégrée avec NumPy et Pandas, et introduit une logique basée sur des tâches qui représente le calcul comme des DAG. Elle permet une configuration personnalisable et la planification du travail sur GPU via Dask-cuDF, mais nécessite des efforts de configuration et d'apprentissage significatifs pour être maîtrisée.

5 Recommandations

Pour orienter efficacement les praticiens vers l'outil adapté à leurs besoins en matière d'optimisation de performances Python, nous présentons la table 1. Elle synthétise les sections du document, mettant en relief leurs caractéristiques clés et contraintes respectives dans le contexte de la Science des Données et de l'Apprentissage Automatique. Afin de refléter la popularité de ces outils, et ainsi avoir une idée de la communauté et du support potentiel associés, la table 2 les répartit selon le nombre d'étoiles de leurs dépôts GitHub respectifs [1]. Nous mentionnons également des noms d'outils et techniques qui ont été étudiés dans (Castro et al., 2023), mais n'ont pas été détaillés dans cet article par souci de concision.

6 Conclusion

Notre article a mis en lumière différentes approches pour améliorer les performances de Python concernant trois scénarios destinés à couvrir la plupart des besoins survenant dans

1. Le nombre d'étoiles a été collecté en Mars 2023

la pratique de la Science des Données. Chaque scénario couvre un stéréotype particulier de problème posé à un *data scientist*. Ils dépeignent la situation d'une utilisation très directe de Python (i.e., en ne recourant qu'à l'API Python standard), l'utilisation de bibliothèques numériques tierces, jusqu'à l'utilisation de frameworks d'apprentissage profond et de solutions de calcul distribué. Malgré la rapide évolution des pratiques de développement logiciel, nous espérons que ce travail donnera une bonne vue d'ensemble et guidera les praticiens dans le choix parmi la pléthore d'outils existants. Outre les praticiens à la recherche d'outils permettant d'accélérer leur code, nous espérons que notre travail pourra aider les nouveaux concepteurs d'outils à obtenir un aperçu de d'état de l'art et des grandes catégories d'outils.

Références

Abadi, M. et al. (2016). TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR abs/1603.04467*.

Bradbury, J. et al. (2018). JAX : composable transformations of Python+NumPy programs. `http://github.com/google/jax`.

Castro, O., P. Bruneau, J.-S. Sottet, et D. Torregrossa (2023). Landscape of High-Performance Python to Develop Data Science and Machine Learning Applications. *ACM Comput. Surv. 56*(3), 1–30.

Coelho, L. P. (2017). Jug : Software for Parallel Reproducible Computation in Python. *Journal of Open Research Software 5*(1).

Dalcín, L., R. Paz, et M. Storti (2005). MPI for Python. *Journal of Parallel and Distributed Computing 65*(9), 1108–1115.

Dask Development Team (2016). *Dask : Library for dynamic task scheduling*. `https://dask.org`.

Ewing, G. et al. (2022). The Cython compiler. `https://cython.org/`.

Guelton, S., P. Brunet, M. Amini, A. Merlini, X. Corbillon, et A. Raynaud (2015). Pythran : Enabling Static Optimization of Scientific Python Programs. *Computational Science & Discovery 8*(1), 014001.

Harris, C. R. et al. (2020). Array programming with NumPy. *Nature 585*(7825), 357–362.

Ismail, M. et G. E. Suh (2018). Quantitative Overhead Analysis for Python. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pp. 36–47.

Kristensen, M. R., S. A. Lund, T. Blum, K. Skovhede, et B. Vinter (2013). Bohrium : Unmodified NumPy Code on CPU, GPU, and Cluster. In *Proceedings of the Workshop on Python for High Performance and Scientific Computing*.

Lam, S. K., A. Pitrou, et S. Seibert (2015). Numba : A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA. Association for Computing Machinery.

Lassner, C. (2015). Pymp. `https://github.com/classner/pymp`.

Moritz, P., R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et I. Stoica (2018). Ray : A Distributed Framework for Emerging AI Appli-

cations. In *USENIX Symposium on Operating Systems Design and Implementation*, pp. 561–577.

Nuitka (2022). Nuitka the python compiler. `https://www.nuitka.net/index.html`.

Nvidia (2022). RAPIDS : Open GPU Data Science. `https://rapids.ai/`.

Pandas Development Team (2022). pandas-dev/pandas : Pandas. `https://github.com/pandas-dev/pandas`.

Paszke, A. et al. (2019). PyTorch : An Imperative Style, High-Performance Deep Learning Library. In *NIPS*, Number 32, pp. 8024–8035.

Pedregosa, F. et al. (2011). Scikit-learn : Machine learning in Python. *Journal of Machine Learning Research 12*, 2825–2830.

Petersohn, D., S. Macke, D. Xin, W. Ma, D. Lee, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, et A. Parameswaran (2020). Towards Scalable Dataframe Systems. In *Proceedings of the VLDB Endowment*, Volume 13, pp. 2033–2046. VLDB Endowment.

Polars (2022). Polars Lightning-fast DataFrame library for Rust and Python. `https://github.com/rapidsai/cudf`.

Ryosuke, O. et al. (2017). CuPy : A NumPy-Compatible Library for NVIDIA GPU Calculations. In *NIPS Workshop on Machine Learning Systems (LearningSys)*.

Sergeev, A. et M. D. Balso (2018). Horovod : Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv preprint arXiv :1802.05799*.

Tejedor, E., Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, et J. Labarta (2017). PyCOMPSs : Parallel Computational Workflows in Python. *The International Journal of High Performance Computing Applications 31*(1), 66–82.

Virtanen, P. et al. (2020). SciPy 1.0 : Fundamental Algorithms for Scientific Computing in Python. *Nature Methods 17*, 261–272.

Álvarez Cid-Fuentes, J., S. Solà, P. Álvarez, A. Castro-Ginard, et R. M. Badia (2019). dislib : Large Scale High Performance Machine Learning in Python. In *Proceedings of the 15th International Conference on eScience*, pp. 96–105.

Summary

Python has become the prime language for application development in the Data Science and Machine Learning domains. However, data scientists are not necessarily experienced programmers. While Python lets them quickly implement their algorithms, when moving at scale, computation efficiency becomes inevitable. Thus, harnessing high-performance devices such as multicore processors and Graphical Processing Units (GPU) to their potential is generally not trivial. In this paper, we convey the main outputs of a recent survey, thought as a reference document for such practitioners to help them make their way in the wealth of tools and techniques available for the Python language. We cast a special focus on delineating main traits and distinctive features in this field. This document can support Data Science practitioners in their tool choice, and tool developers in the identification of potential lacks in existing work.