Élements Logiciel pour le Traitement de Données Massives

Parallélisation de la Fonction GEMM par le Framework BLIS

César Roaldès & Morgane Hoffmann

Février 2020

1 Introduction

L'objectif de ce projet est d'implémenter le framework d'algèbre linéaire BLIS (BLAS-like Library Instantiation Software), afin de mesurer la vitesse d'execution du produit matriciel sur différentes tailles de matrices. Cette opération, à la base des calculs scientifiques, se verra appellée depuis de langage Python. Notre travail prendra pour référence l'article [1] *"Anatomy of High-Performance Many-Threaded Matrix Multiplication"* dans lequel les auteurs exploitent les nouvelles opportunités de parallélisation rendues accessibles par BLIS. Nous commencerons par présenter les différents avantages apportés par la librairie avant d'aborder l'étape délicate de son installation ainsi que l'installation d'une version de NUMPY l'exploitant. Nous consacrerons finalement la dernière partie à la présentation des résultats obtenus sur nos ordinateurs.

2 Les spécificités de BLIS

BLIS reprend l'approche d'implémentation initiée par OpenBLAS, l'étape de confirguration détecte les caractéristiques des composants de la machine sur laquelle l'API est installée. En particulier, l'analyse de la taille des registres et des différents caches du processeur est cruciale si l'on souhaite maximiser les performances de la librairie en stockant les plus grands blocs de matrice possibles dans les différents caches CPU.

La fonction de produit matriciel appellée *gemm* (General Matrix Multiplication) implémentée par OpenBLAS repose sur l'exploitation d'un *microkernel* contenu dans une triple boucle. BLIS propose d'ouvrir ce microkernel pour y introduire deux nouvelles boucles en son coeur, rendant l'API intéressante dans un contexte de parallélisation, et plus particulièrement lorsque ces calculs sont effectués sur une architecture manycore. Le microkernel représente la plus petite unité de calcul intervenant dans la fonction gemm, dont la version proposée par BLIS est intégrée dans une quintuple boucle.

D'autres spécificités, qui ne seront pas développées dans notre devoir mais représentent un apport notable de BLIS, sont :

— La généralisation du stockage mémoire des matrices (stockage en lignes ou par colonnes).

— La gestion des calculs intégrant des nombres complexes.

— La multiplication de matrices dont types sont différents

3 Implémentation de BLIS

Nous compilons BLIS directement depuis son code source en libre accès sur GitHub [1] et a été réalisée sur le système d'exploitation Ubuntu 16.04. L'installation de la librairie se fait en trois étapes :

1. **Détection des composants et définition des options d'installation.**

   A l'instar d'OpenBLAS, les caractéristiques et l'architecture du CPU sont détectées par une procédure automatisée (ou spécifiées au préalable) lors de l'étape de configuration. C'est la microarchitecture *haswell* qui a été détectée sur notre ordinateur. L'option d'activation de la parallelisation des calculs à l'aide d'`openmp` est également spécifiée, `--enable-threading=openmp`. La parallelisation par la librarie `pthread` est également disponile mais déconseillée par les développeurs. De plus, l'utilisation de la commande `--enable-cblas` permet d'incorporer BLIS dans une interface CBLAS, rendant ainsi la librairie candidante à une utilisation depuis NUMPY.

2. **Compilation des fichiers source**

   L'étape de compilation des fichiers sources se fait avec le logiciel `make`. Cette opération fait intervenir un Makefile configurée lors de l'étape précédante. Le header file `"blis.h"` et la librairie partagée `libblis.so` sont ainsi générés, cependant, cette opération n'exécute qu'une compilation est la librairie n'est accessible que localement.

3. **Installation de la librairie**

   La dernière étape est l'installation, cette étape permet d'utiliser BLIS en rendant accessible son header file et la librairie partagée depuis n'importe quel dossier de notre ordinateur.

Notre objectif étant d'utiliser BLIS depuis python, la librairie `cython-blis`, qui l'exploite, semblait toute indiquée. Cependant, ce package ne permet pas de réaliser des calculs de manière prallélisée et a donc été mise de côté dans

1. `https://github.com/flame/blis`

le cadre de ce projet. C'est par le biais de NUMPY, qui permet de choisir parmis une large gamme la librarie d'algèbre linéaire optimisée de BLAS, que nous utilisons BLIS. Cette dernière n'est pas proposé par défaut et la configuration doit s'effectuer depuis le fichier `site.cfg` [2] dans lequel on spécifie la localisation de la librarie de BLIS mais utilisons son interface CBLAS installée via l'option `--enable-cblas` en pointant la location du header vers le répetroire du fichier `cblas.h` créé lors de la compilation. La compilation de cette version du package NUMPY reposera alors sur une utilisation transparente de BLIS pour l'utilisateur, comme l'est celle de MKL ou OpenBLAS.

4 Expérimentations

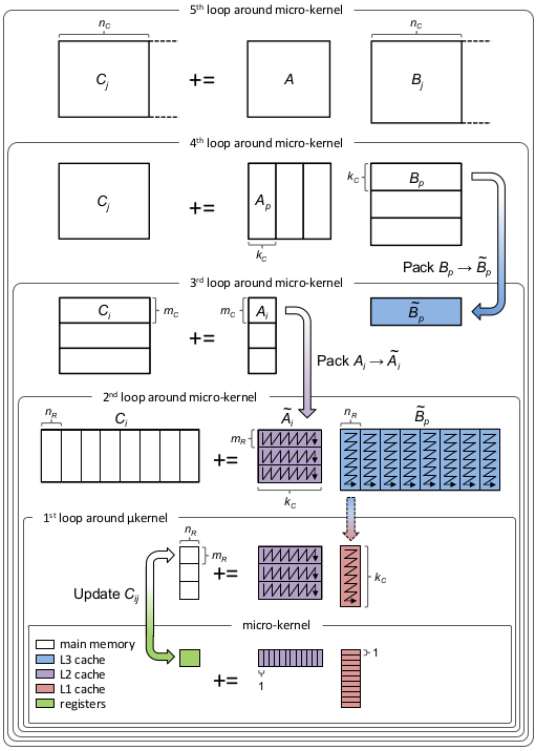


Figure 1 – Boucles de GEMM
(Source : `blis/docs/Multithreading.md`)

L'expérimentation suivante, et les résultats en découlant, a été réalisée sur un processeur Intel Core i5-9600K CPI @ 3.70GHz. Ce processeur dispose de 6 coeurs partageant un unique socket. Cette concentration de tout les coeurs sur le même socket permet le partage du cache L3 (9216 Ko), tandis que les caches L2 (256 Ko) et L1 (32 Ko) sont privés. Le nombre de thread par coeur de ce processeur est limité à 1, ce qui restreint notre nombre d'opération executables en parallèle à 6, et avec lui la diversité des stratégies de parallélisation réalisables.
Il existe trois méthodes de spécification de la stratégie de parallélisation. La première se fait de manière globale par la configuration (préalable à l'exécution des scripts) des variables d'environnement. Les deux autres sont spécifiées directement depuis le code source des scripts appellant la librairie, et la seule utilisation faite de l'une de ces deux dernières [3] a été utilisée dans la fonction C présentée à titre de comparaison avec les performances obtenues en Python.

Pour chacune de ces trois méthodes il existe deux façons de spécifier la stratégie de parallèlisation. La manière automatique, qui indique simplement le nombre de coeurs utilisés, et la manière manuelle. La stratégie de parallèlisation de manière automatique se fait à l'aide de techniques heuristiques, tandis que la spécification manuelle permet d'indiquer quelles sont les boucles qui seront amenées à être parallelisées et le degré de parallélisation de chacune d'entre elles. La méthode manuelle est donc à privilégier lorsqu'on souhaite maîtriser la stratégie de parallélisation.

La figure 1 indique les cinq boucles de la fonction gemm. Parmis ces boucles, seules la 1ère, la 2nd, la 3ème et la 5ème sont candidates à la parallélisation [4]. Les variables d'environnement spécifiant la stratégie de parallélisation de ces boucles sont regroupées dans la table 1.

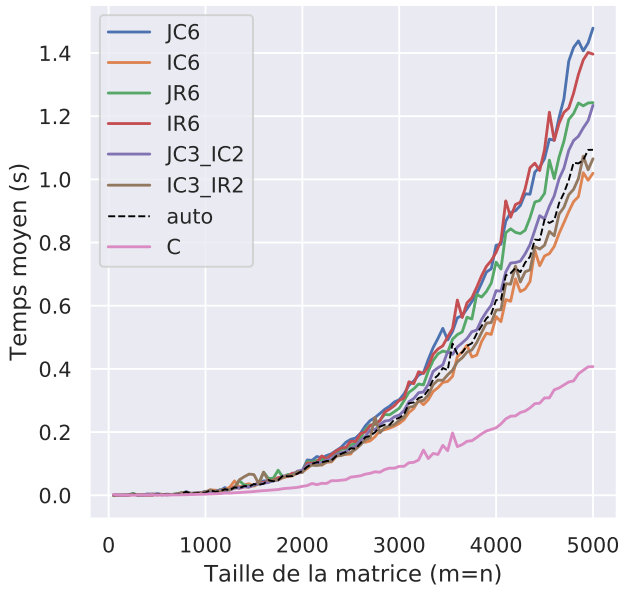
5ème boucle	`BLIS_JC_NT`
3ème boucle	`BLIS_IC_NT`
2ème boucle	`BLIS_JR_NT`
1ère boucle	`BLIS_IR_NT`

Table 1 – Variables d'environnement globales spécifiques à chaque boucle

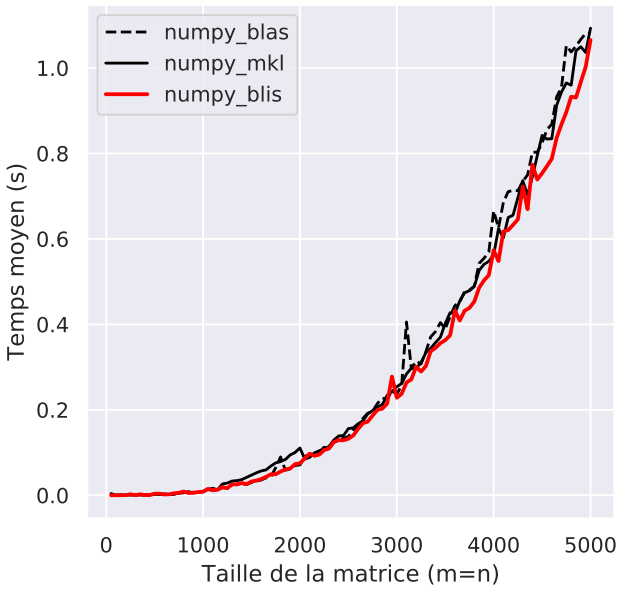
La figure 2a présente le temps d'exécution moyen de plusieurs stratégies de parallélisation de la fonction gemm de blis depuis Python et C [5]. La légende indique les boucles parallélisées ainsi que leur degré de parallelisation. Les performances de la stratégie automatique sont également affichées et obtient des résultats proches des meilleures startégies de parallélisation définies manuellement. Les résultats obtenu à partir de nos multiples implémentations manuelles sont en accord avec les résultats théoriques avancés par les développeurs. Ces derniers préconisent une parallélisation sur la boucle IC (3ème boucle) lorsque seul le cache L3 est partagé, comme dans notre cas, et c'est en effet cette stratégie de parallélisation qui nous donne les meilleurs résultats (courbe orange). La machine sur laquelle nous travaillons ne possédant pas de mémoire partagée sur les caches L2 et L1, l'exploitation des boucles JR et IR n'est pas efficace, et l'augmentation du nombre de threads au delà de 6 ralentit de manière considérable le processus.

Suite aux bonnes performances de la spécification *IC6*, nous reprennons cette dernière afin de comparer la vitesse de calcul de la fonction gemm de BLIS aux vitesses de calcul obtenues par deux autres bibliothèques "BLAS-like", OpenBLAS et MKL depuis le package NUMPY. Les performances des 3 librairies sont présentées dans la figure 2b. La librarie BLIS obtient de meilleures performances que les benchmarks que nous avons choisi. Cela est surprenant car des librairies comme MKL sont ultra-optimisées pour les processeurs Intel. Il convient de noter que les améliorations de performance obtenues restent modestes et que les

2. Qui doit être dans le même répertoire que setup.py lors de la compilation de NUMPY.
3. Locally at runtime - The manual way
4. La 4ème boucle n'est pas encore disponible
5. Ce graphique est construit en calculant les temps d'execution des différentes méthodes pour des matrices carrées de même taille m de différentes tailles. On augmente m de 50 en 50 et on reitère à chaque étape les calculs 5 fois. Les résultats présentés sont donc des moyennes de temps d'execution.



(a) Stratégies de parallélisation BLIS



(b) Comparaison des différentes API depuis NUMPY

Figure 2 – Temps d'execution de la fonction GEMM

vitesses de calculs sont comparables, dans le cas du produit matriciel, pour les trois librairies. Il serait intéressant d'effectuer le même exercice pour d'autres opérations de niveau 3.

Conclusion

Ce devoir nous aura permis de découvrir plusieurs aspects fondamentaux en informatique mais qui nous étaient jusqu'ici inconnu. Parmis eux, la découverte des algorithmes d'algèbre linéaire sur lesquels reposent les célèbres packages Python NUMPY/SCIPY que nous utilisons quotidiennement. Nous avons également découvert le système d'exploitation Linux et les serveurs AWS, mais également découvert l'utilisation des variables d'environnement et les librairies C, à apprendre à lire les documentations, installer des programmes depuis leurs fichiers sources, ou bien encore à utiliser les avantages qu'offrent Github. Pour finir, le sujet que nous avons choisi nous a amené à nous intéresser à l'architecture et micro-architecture du processeur ainsi qu'aux manières d'exploiter l'augmentation du nombre de coeur sur nos CPU grâce à la parallélisation. Nous aurions cependant aimé essayer d'autres stratégies de parallélisation plus complexes sur des ordinateurs plus puissants le permettant et réaliser un benchmark sur d'autres proposée par BLIS et d'autres applications plus complexes.

Bibliographie

[1] Tyler M. Smith, Robert A. van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. Anatomy of high-performance many-threaded matrix multiplication. pages 1049–1059, 2014.