RÉPUBLIQUE DU CAMEROUN
****
Paix - Travail - Patrie
****
UNIVERSITÉ DE YAOUNDÉ I
****
FACULTÉ DES SCIENCES
****
DÉPARTEMENT D'INFORMATIQUE
****

REPUBLIC OF CAMEROON
****
Peace - Work - Fatherland
****
UNIVERSITY OF YAOUNDE I
****
FACULTY OF SCIENCE
****
DEPARTMENT OF COMPUTER SCIENCE
****

CARACTÉRISATION ET MODÉLISATION D'ARCHITECTURES ARM 64 BITS TX2 POUR DES APPLICATIONS ET ALGORITHMES DE RECOMMANDATION

En vue de l'obtention du diplôme de Master Recherche en Informatique

Option : Science de Données

Présenté par :
NANA TCHAKOUTÉ Roblex

Matricule :
16U2639

Co-Directeur du mémoire : Pr Jean-François MÉHAUT (LIG, UGA)
Co-Directeur du mémoire : Pr Maurice TCHUENTE (Dept Info, UYI)

Année académique 2021/2022

Université de Yaoundé I
Faculté des Sciences
Département d'Informatique
BP 812 Yaoundé, Cameroun

Laboratoire d'Informatique de Grenoble
Bâtiment IMAG-700 avenue Centrale
CS 40700-38058 Grenoble Cedex

UMMISCO
IRD, Université de Sorbonne
F-93 143, Bondy France

FR2I
BP 14306 Yaoundé,
Cameroun

Dédicaces

Je dédie ce travail à ma famille.

Remerciements

La réalisation de ce mémoire a été possible grâce à plusieurs personnes à qui je voudrais témoigner toute ma gratitude. Ainsi, ma reconnaissance s'adresse aux personnes suivantes :

Au **Pr. Jean-François MÉHAUT**, à qui j'adresse ma gratitude pour la qualité de l'encadrement, pour la patience, la rigueur, le professionnalisme et la disponibilité dont il a toujours fait preuve. J'adresse une immense reconnaissance pour ce sujet passionnant autour de l'architecture qui m'a fait découvrir d'autres horizons de l'informatique.

Au **Pr. Maurice TCHUENTE**, pour m'avoir sélectionné, encadré, orienté, encouragé et conseillé ; pour la méthodologie de recherche qu'il m'a enseignée ; et pour l'école de vie à travers ses histoires inspirantes. Je suis infiniment reconnaissant pour ce parcours de master 2.

Aux membres du jury pour leur entière disponibilité et pour l'honneur qu'ils m'accordent en acceptant d'évaluer ce travail.

Aux **Dr. Thomas MESSI** et **Dr. Armel NZEKON** pour leurs encouragements, leurs suggestions et leurs conseils qui m'ont permis d'avancer dans ce travail.

À l'Université de Yaoundé I notamment au Département d'Informatique pour la qualité de ses enseignements durant mon parcours jusqu'à présent.

À l'équipe **UMMISCO** au travers de son laboratoire, pour le cadre de travail mis à notre disposition ; au laboratoire **LIG** de Grenoble au travers du Pr. Jean-François MÉHAUT qui nous a fourni l'accès à la plateforme **GRID5000** pour les expérimentations et à la fondation **FR2I** pour les enseignements en méthodologie de recherche.

À ma famille, et tout particulièrement ma très chère Maman, pour son amour, ses encouragements, ses prières, son soutien sans faille ; mon oncle Mr. Silvain pour son soutien infini depuis des années ; à tous mes frères et sœurs pour leur amour et confiance sans pareil.

Pour terminer, je remercie mes partenaires et camarades de promotion **Roméo KOATI** et **Audrey DONGMO** pour notre travail en équipe qui porte ses fruits ; **Vanessa FOKOU, Gwladys KELODJOU, Brice TCHUENKAM, Antoine TSAGMO, Franklin DONGMO** et **Dimitri KAMDEM** pour nos échanges et leurs relectures.

À toutes ces personnes ayant contribué de près ou de loin à la réalisation de ce travail, je présente ma profonde gratitude.

Resumé

L'économie d'énergie est devenue l'un des principaux défis pour les nouvelles générations de serveur et super-calculateurs. Certains constructeurs de systèmes de calcul haute performance (**HPC**) trouvent une solution en se basant sur des composants basse consommation **ARM** aujourd'hui présents dans la grande majorité des systèmes embarqués ou mobiles. En effet, la particularité de ces composants est leur faible consommation énergétique pour des performances compétitives avec les architectures **Intel** bien connues du domaine. Plusieurs travaux ont été réalisés sur ces architectures ARM pour l'évaluation des performances sur des applications de calcul scientifique. Dans ces travaux, les quantités de données transférées entre les niveaux de la hiérarchie mémoire sont connues à l'avance, car les accès mémoire sont prévisibles et ne dépendent que de la taille du problème. Malheureusement, ce n'est pas le cas pour les applications où les modèles d'accès à la mémoire dépendent des données qui ne sont connues qu'au moment de l'exécution. C'est notamment le cas pour les applications de recommandation à base de graphes, où les schémas d'accès à la mémoire sont irréguliers et non-prévisibles, et pour lesquels les performances des systèmes HPC sont souvent limitées par le sous-système mémoire. Notre travail porte sur la modélisation de la performance des architectures ARM 64 bits **TX2** pour des algorithmes de recommandation à base de graphes. Nous présentons une méthode simple permettant de réaliser de bonnes estimations de performance et de déterminer l'accélération maximale que peut induire la parallélisation. Comme étude de cas, nous considérons deux applications **GraFC2T2** et **PageRank** qui ont la multiplication matrice creuse-vecteur comme noyau irrégulier. En utilisant deux systèmes de CPU multi-cœurs et un ensemble de jeux de données issues des sites ouverts à l'exemple de stanford.edu pour les réseaux sociaux et les sites web, nous montrons que la plupart des modèles de performance proposés, quantifient avec précision les goulots d'étranglement pour les applications étudiées. Les résultats montrent aussi que les architectures Intel sont généralement plus performantes que les architectures ARM.

Mots-clés : microarchitecture, parallélisation, analyse de performance, processeur ARM, accès-mémoire irrégulier, multiplication matrice creuse-vecteur, PageRank.

Abstract

Energy saving has become one of the main challenges for new generations of servers and supercomputers. Some manufacturers of high-performance computing systems (**HPC**) find a solution based on low-power **ARM** components present today in the vast majority of embedded or mobile systems. Indeed, the particularity of these components is their low energy consumption for competitive performance with well-known **Intel** architectures in the field. Several works have been carried out on these ARM architectures for performance evaluation on scientific computing applications. In these works, the amounts of data transferred between the levels of the memory hierarchy are known in advance, because the memory accesses are predictable and depend only on the size of the problem. Unfortunately, this is not the case for applications where memory access patterns depend on data that is only known at runtime. This is particularly the case for graph-based recommendation applications, where memory access patterns are irregular and unpredictable, and for which the performance of HPC systems is often limited by the memory subsystem. Our work focuses on modeling the performance of 64-bit ARM **TX2** architectures for graph-based recommendation algorithms. We present a simple method allowing to make good performance estimates and to determine the maximum acceleration that parallelization can induce. As a case study, we consider two maps **GraFC2T2** and **PageRank** which have sparse matrix-vector multiplication as their irregular kernel. By using two multi-core CPU systems and a set of data sets from open sites such as stanford.edu for social networks and websites, we show that most of the proposed performance models quantify with pinpoint the bottlenecks for the applications studied. The results also show that Intel architectures generally perform better than ARM architectures.

Keywords : microarchitecture, parallelization, performance analysis, ARM processor, irregular memory access, sparse matrix-vector multiplication, PageRank.

Table des matières

Table des figures

Liste des tableaux

Liste des abréviations

1. **ARM** : **A**dvanced **R**isc **M**achine
2. **CISC** : **C**omplex **I**nstruction **S**et **C**omputing
3. **CMP** : **C**hip **M**ultiprocessor
4. **COO** : **C**oordinate format
5. **CPU** : **C**entral **P**rocessing **U**nit
6. **CSR** : **C**ompressed **S**parse **R**ow
7. **DDR4** : **D**ouble **D**ata **R**ate version 4
8. **DRAM** : **D**ynamic **R**andom **A**ccess **M**emory
9. **ECM** : **E**xecution **C**ache **M**odel
10. **FLOPS** : **FL**oating **P**oint **O**peration per **S**econd
11. **HPC** : **H**igh **P**erformance **C**omputing
12. **ISA** : **I**nstruction **S**et **A**rchitecture
13. **MTS** : **M**ega **T**ransfer per **S**econd
14. **NUMA** : **N**on **U**niform **M**emory **A**ccess
15. **OI** : **O**perational **I**ntensity
16. **OS** : **O**perating **S**ystem
17. **RISC** : **R**educe **I**nstruction **S**et **C**omputing
18. **SMM** : **S**imple **M**emory **M**odel
19. **SRAM** : **S**tatic **R**andom **A**ccess **M**emory
20. **SpMV** : **S**parse **M**atrix **V**ector **M**ultiplication
21. **TX2** : **T**hunder**X**2

Introduction

Dans l'industrie du *HPC(High Performance Computing)*, on assiste ces dernières années à une période de grands changements. Ces changements provoqués en grande partie par la fin de la loi de **Moore** qui prédisait que le nombre de transistors sur une plaque de silicium va doubler tous les dix-huit(18) mois. Cela est dû en partie à la difficulté des fonderies à réaliser des transistors de plus en plus petits. Ce qui a poussé l'industrie à améliorer les performances des processeurs par l'ajout des fonctionnalités sur la microarchitecture. Ainsi, les architectures récentes intègrent plusieurs composantes, à l'instar des *mémoires caches, unités de préchargement, des pipelines, etc.*. Cette variabilité au niveau architectural ouvre des opportunités pour les nouveaux entrants sur le marché du *HPC* à savoir les constructeurs à base d'architectures *ARM(Advanced Risc Machine)*, afin de pouvoir fournir des architectures plus spécialisées pour des charges de travaux particulières. Les architectures *ARM* sont très populaires et ont été majoritairement utilisées dans les téléphones mobiles pendant des années. La motivation d'utiliser ces architectures pour le *HPC* vient de leur compétitivité en termes de performance à ceux d'Intel pour une faible consommation énergétique. Les travaux de [Hammond et al., 2019] ont démontré que les architectures ARM Thunder X2 peuvent être plus performantes que celles d'Intel de la même génération pour certaines applications dominées par les accès mémoire. Ainsi, on cherche à comprendre le comportement de ces processeurs ARM pour les applications de recommandation.

La performance est une priorité élevée dans les serveurs et les supercalculateurs, et un travail méticuleux est donc consacré à l'optimisation du code sous-jacent. Au cours de ces efforts d'optimisation, les modèles de performance sont des outils précieux pour diriger l'attention vers les points de pression et indiquer quand les optimisations sont suffisantes et qu'il serait improductif de déployer des efforts supplémentaires. Par exemple, le célèbre modèle **Roofline** [Williams et al., 2009] limite les performances en termes de capacité de calcul maximale et de bande passante mémoire d'un processeur, ainsi que d'intensité de calcul d'un algorithme. Récemment, plusieurs travaux ont été réalisés dans la littérature autour de cette question de performance des applications sur des architectures. On peut parler des travaux de [Pourroy, 2020] qui s'est concentré à l'étude et à la modélisation des performances des applications *HPC* de calcul scientifique ainsi que ceux de [Hammond et al., 2019] et [Calore et al., 2020] qui une fois de plus se concentrent sur des applications de calcul scientifique, mais cette fois-ci avec les architectures *ARM TX2(Thunder X2)*. Ces travaux ont permis de montrer que les architectures *TX2* peuvent être assez compétitives à celles d'*Intel* pour certaines applications de calcul scientifique. Dans ces travaux, les quantités de données transférées entre les niveaux de la hiérarchie mémoire sont connues à l'avance, car les accès mémoire sont prévisibles et ne dépendent que de la taille du problème. Malheureusement, ce n'est pas le cas pour les calculs irréguliers, où les modèles d'accès à la mémoire dépendent de données qui ne peuvent être connues qu'au moment de l'exécution. Ce qui rend ces accès imprévisibles et les performances sont plus difficiles à prédire.

Face à ces schémas d'accès irréguliers comme nous l'avons vu avec les applications de recommandations basées sur les graphes, l'approche typique consiste à dériver des estimations du trafic mémoire pour les scénarios les plus défavorables ou les plus favorables. Il s'agit d'estimations **"papier-crayon"** qui ont l'avantage d'être peu coûteuses à produire, ne nécessitant aucune implémentation ou machine réelle à exécuter. En général, de nombreux noyaux de calcul sont confrontés

aux mêmes problèmes en raison des accès irréguliers à la mémoire qui résultent de l'utilisation de structures de données éparses, telles que des graphes ou des matrices creuses.

Dans ce travail, nous nous concentrons sur le problème de prédiction des performances des applications de recommandation à base de graphe en terme de trafic mémoire et de temps d'exécution. Donc à la différence des travaux de la littérature, notre étude ne porte pas sur les applications *HPC* de calcul scientifique. Ainsi, nous procédons d'abord par une implémentation de ces applications en *C/C++*. Ensuite, nous proposons une méthodologie qui réutilise quelques modèles et outils de la littérature qui sont les plus représentatifs de notre problème. Contrairement à la méthodologie de [Pourroy, 2020] qui permet de choisir la meilleure architecture pour porter et optimiser son code, notre méthodologie est utilisée pour la prédiction et la comparaison des performances. Elle est appliquée dans ce mémoire aux architectures **Intel Skylake 6130** et **ARM Thunder X2**. Plus précisément, les expérimentations ont été menées sur les serveurs de la grille de calcul **GRID5000** [1].

Le reste du manuscrit suit le plan suivant : le chapitre 1 présente le background technologique ou contexte technologique de nos travaux ainsi que des généralités sur les architectures des processeurs *ARM TX2* avec un accent sur la comparaison des architectures *TX2* et *Intel*. Le chapitre 2 présente un état de l'art sur les modèles de performances de la littérature. Dans le chapitre 3, nous présentons nos contributions sur l'analyse des applications de recommandation par la mise en œuvre de ces applications et une méthodologie utilisant les modèles de performance les plus adaptés. Dans le chapitre 4, nous faisons une étude expérimentale afin d'utiliser les prédictions des modèles et d'interpréter les résultats d'exécutions réelles par des comparaisons. Nous terminons par une conclusion et quelques perspectives.

1. https ://www.grid5000.fr/w/Grid5000 :Home

Chapitre 1: Background technologique

1.1 Généralités sur l'architecture des ordinateurs

Pour bien réaliser les travaux de caractérisation et d'analyse de performances, il est nécessaire d'avoir de solides connaissances des éléments de l'architecture. Dans cette section, nous présentons une vue globale des fonctionnalités clés des processeurs modernes qu'il est indispensable de connaître pour comprendre la performance des applications. Nous étudions ensuite la hiérarchie mémoire qui est la ressource critique de nombreuses applications.

1.1.1 Description des architectures

Nous commençons cette section par la définition de la notion d'architecture dans le cadre de notre mémoire et en faisant la différence avec deux termes qui prêtent à confusion dans la littérature.

ISA

L'**ISA** ou tout simplement **jeu d'instructions** est l'ensemble des instructions machines qu'un processeur est capable d'exécuter. Ces instructions machine permettent d'effectuer des opérations élémentaires (*opérations arithmétiques et logiques*) ou plus complexes (adressage mémoire, passage en mode basse consommation, etc.). C'est une spécification formelle qui peut être utilisée par plusieurs fabricants de microarchitecture. Le jeu d'instructions précise aussi quels sont les registres du processeur manipulables par le programmeur (les registres architecturaux). Il sert d'interface entre la couche matérielle et la couche logicielle et assure la compatibilité de programme sur des microarchitectures de différents constructeurs. Plusieurs *ISA* existent, mais les plus populaires sont les *ISA* **x86**, **AMD64**, **RISC-V** et **ARM**. Généralement, on les regroupe en deux grandes familles : **RISC(Reduce Instruction Set Computer)** et **CISC(Complex Instruction Set Computer)**.

Les puces ARM qui font partie de la famille RISC ne supportent que des instructions simples et de taille fixe (4 octets pour l'ISA ARM standard), s'exécutant en un nombre constant de cycles, à l'inverse des puces x86 qui proposent des instructions nécessitant plus de cycles que d'autres pour réaliser certaines tâches complexes. Les puces ARM supportent également moins de modes d'adressage (la façon de référencer une donnée dans une instruction), la plupart des instructions ne pouvant travailler qu'avec des données présentes dans un registre processeur, alors que la majorité des instructions x86 peuvent aller chercher des données directement en mémoire. L'intérêt d'une architecture RISC se situe principalement dans la moindre complexité du schéma électronique de la puce : pour atteindre un même niveau de performances, il faut généralement moins de transistors, ce qui permet de réduire les coûts de production et d'avoir une meilleure efficacité thermique. Du fait de l'obligation de faire des accès mémoire explicites dans le code, les processeurs RISC sont également souvent dotés d'un plus grand nombre de registres, ce qui permet de réduire la fréquence des accès mémoire. Ainsi, l'architecture ARMv8 en comporte 31 [1], alors que le x86 n'en possède que 8 pour le code 16/32/64 bits et 8 supplémentaires réservés au code 64 bits [2].

1. source : https ://developer.arm.com/documentation/dui0204/j/CEGIBCCG
2. source : http ://resource.renesas.com/lib/eng/e_learnig/h8_300henglish/s04/bf01.html

Les architectures CISC ont pour leur part l'avantage de permettre un code exécutable plus compact, réduisant ainsi le nombre d'accès à la mémoire. Par ailleurs, même si elles sont plus longues à exécuter que les instructions simples, certaines instructions spécialisées peuvent s'exécuter bien plus vite que leur équivalent en instructions simples. Notons toutefois qu'en pratique, les processeurs x86 d'aujourd'hui ont en fait un cœur d'exécution interne plus proche d'une architecture RISC que d'une architecture CISC, associé à un étage de décodage des instructions qui s'occupe de convertir le flux d'instructions x86 en instructions internes plus simples. Ces processeurs restent des processeurs CISC, du fait de leur interface interne, mais le recours à une architecture interne RISC permet de réduire la complexité des unités d'exécution. À l'inverse, les processeurs ARM (et bon nombre d'autres processeurs RISC) ont adopté des instructions spécialisées, notamment pour les traitements multimédias, via des extensions du jeu d'instructions de base, comme par exemple l'extension *NEON*.

Malgré des philosophies initiales très différentes, les puces ARM et x86 sont donc de plus en plus proches techniquement : les puces ARM haut de gamme se complexifient pour gagner en performances, atteignant désormais les performances de l'entrée de gamme x86, pendant que les puces x86 d'entrée de gamme se simplifient pour gagner en consommation, au point que les plus petits cœurs x86 d'aujourd'hui commencent à approcher la taille des gros cœurs ARM. Par exemple, avec une même finesse de gravure de 28 nm, un cœur AMD Jaguar occupe 3.1mm² contre 1.62mm² pour un ARM Cortex-A15 offrant des performances du même ordre. Un cœur Haswell gravé en 28 nm occuperait pour sa part une surface de l'ordre de 23mm², mais avec des performances sans commune mesure avec celles d'un Cortex-A15 [3].

Sur les processeurs *TX2*, on a l'*ISA ARMv8* qui fait partie de la famille *RISC* alors que les architectures *Intel* sont basées sur l'*ISA x86* ou *AMD64*. Ainsi, ces processeurs sont en réalité plus proches que ce qu'on imagine au regard des familles d'ISA.

Microarchitecture

La **microarchitecture** est l'implémentation matérielle d'un *jeu d'instructions*. Elle regroupe l'ensemble des composantes matérielles permettant le traitement dans le processeur (pipeline, unité de préchargement, mémoires caches, etc.). Si l'*ISA* est considéré comme la couche logicielle de l'architecture, alors la microarchitecture est la couche matérielle. Pour des besoins de performance, il est souvent nécessaire d'avoir connaissance de la microarchitecture. La conception d'une microarchitecture commence par le choix de *l'ISA* à implémenter. Les différences principales entre deux microarchitectures implémentant le même *ISA* sont leurs différences de performances et de coût [Pourroy, 2020].

Nous utilisons le terme architecture pour faire référence à une microarchitecture qui implémente une *ISA*. Par exemple, l'architecture *TX2* fait référence à la microarchitecture **Vulcan** [4] impémentant l'*ISA* **ARMv8**. La figure 1.1 montre un exemple de représentation graphique des éléments de la microarchitecture *Vulcan*.

Description de la microarchitecture Vulcan de TX2.

Vulcan est une microarchitecture superscalaire (i.e qui a au moins deux pipelines) avec un moteur d'exécution dans le désordre (i.e dans un ordre différent de celui du programme en exécution) qui prend en charge jusqu'à quatre *threads* matériels simultanément. Vulcan possède un pipeline de 13 à 15 étages, légèrement plus court que celui d'Intel de la même génération Skylake, qui est de 14 à 19 étages. Cela donne l'avantage à Intel pour plus de parallélisme d'instructions.

3. source : https ://www.infobidouille.com/pcworld/la-question-technique-17-arm-contre-x86-quelles-differences-fondamentales/

4. https ://en.wikichip.org/wiki/cavium/microarchitectures/vulcan

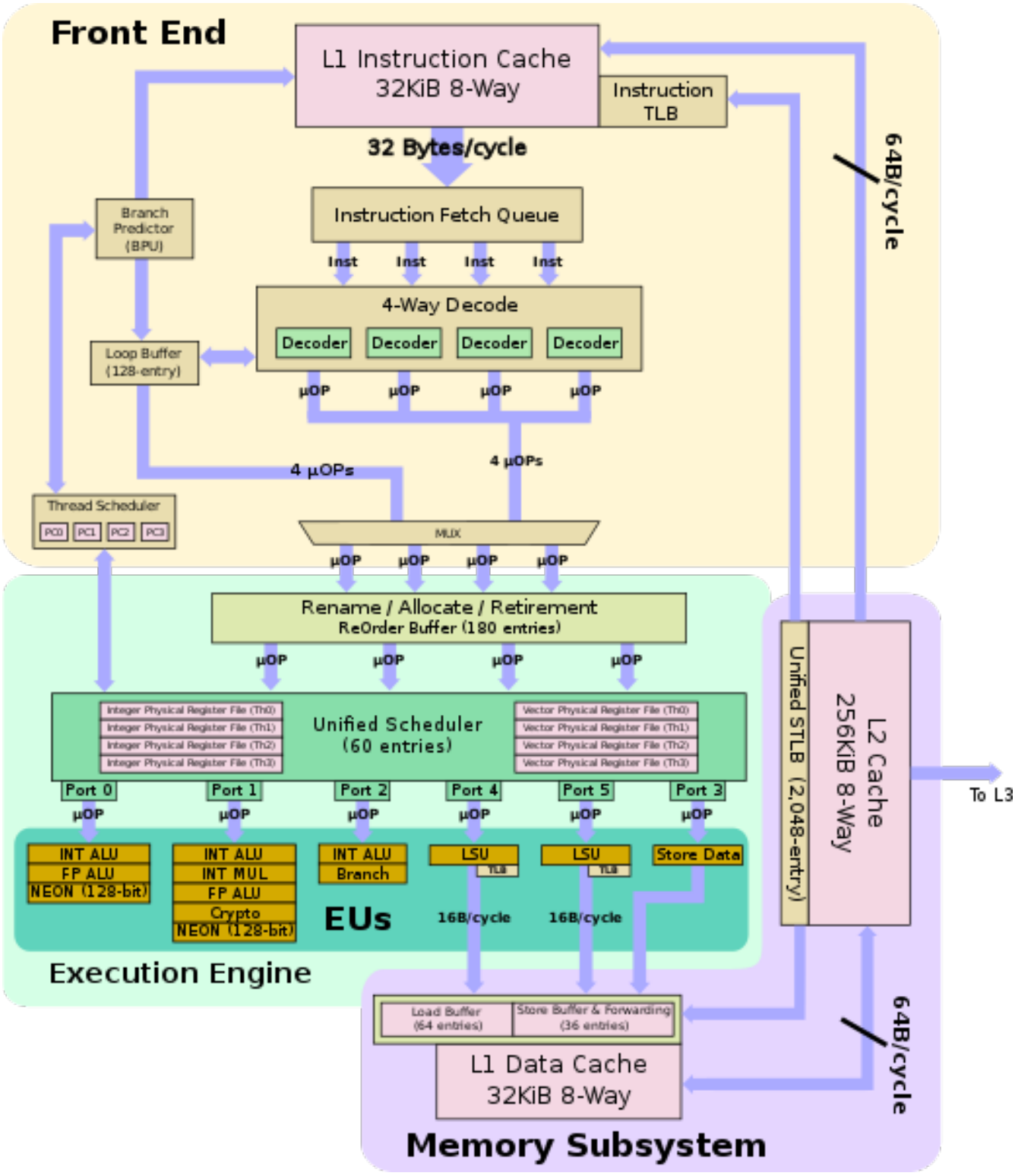

FIGURE 1.1 – Microarchitecture *Vulcan* de *TX2* sous forme de diagramme de bloc d'un cœur de processeur. On y retrouve les principaux éléments de la microarchitecture ainsi que la schématisation du pipeline d'exécution. Source : en.wikichip.org

Le **front-end** de Vulcan a pour tâche de récupérer les instructions d'un flux d'instructions de *threads* prêts à l'emploi et de les introduire dans le décodeur afin de les transmettre aux unités

d'exécution. Vulcan supportant jusqu'à quatre threads, un planificateur de threads détermine à partir de quel flux d'instructions de threads opérer. Cette détermination est effectuée à chaque cycle avec l'aide du prédicteur de branchement, sans coût supplémentaire. Ainsi, ce mécanisme permet d'utiliser efficacement les unités d'exécutions du cœur. Ce mécanisme n'est pas présent sur Intel d'autant que les processeurs Intel ont maximum deux threads simultanés.

Vulcan dispose d'un cache L1 de 32 KiB, associatif à 8 voies, et d'une TLB dédiée aux instructions L1. Il est intéressant de noter que tous les caches centraux de Vulcan sont associatifs à 8 voies afin d'aider le prédicteur de branchement qui travaille sur des modèles de *stride* (sauts d'adresse mémoire) de ligne de cache. Dans le cadre d'un flux normal, les données à extraire ont déjà été prédites et sont arrivées dans le L1 depuis le L2.

La récupération des instructions se fait sur une fenêtre de 32 octets ou 8 instructions ARM (4 octets). Ce débit est deux fois supérieur à celui de l'architecture Skylake. Le flux d'instructions est décomposé en ses instructions constitutives où elles sont mises en file d'attente pour aller sur le décodeur. La file d'attente est partagée par tous les threads. Malheureusement, taille de la file d'attente n'a pas été divulguée.

À chaque cycle, jusqu'à quatre instructions sont envoyées au décodeur. Dans la conception précédente, les produits de *Broadcom* décodaient les instructions MIPS [5]. Avec Vulcan, le passage à ARM a nécessité le remplacement du décodeur par une logique beaucoup plus complexe qui décode l'instruction d'origine et émet des **µOPs (micro-opérations)**. Dans la plupart des cas, il existe une correspondance 1 :1 entre les instructions et les µOPs, avec une moyenne de 15% de µOP supplémentaires émis par les instructions. La complexité supplémentaire a ajouté un autre étage de pipeline au décodage. On peut noter une philosophie similaire sur Skylake. Cependant, Skylake bénéficie d'une implémentation de Cache **"L0"** pour stocker les µOPs.

Entre le décodeur et l'ordonnanceur, se trouve un tampon de boucle de 128 entrées. Le tampon de boucle, en conjonction avec le prédicteur de branchement, mettra en file d'attente les opérations de boucle serrée récentes. Le tampon lira les opérations de manière répétée jusqu'à ce qu'une prise de branche se produise. Lorsque cela se produit, le front-end (récupération d'instructions, décodage, etc.) est en grande partie mis hors tension afin d'économiser de l'énergie. Un mécanisme similaire se trouve sur Skylake pour désactiver le *"Front-end"*.

Le **"back-end"** de Vulcan gère l'exécution des opérations dans le désordre. Le back-end de Vulcan a été considérablement amélioré par rapport aux conceptions précédentes, notamment par une refonte complète de l'ordonnanceur. La plupart des améliorations ont consisté à retravailler entièrement l'ordonnanceur afin d'extraire plus efficacement les opportunités de parallélisme au niveau des instructions. À partir du décodage, les instructions sont envoyées au *Reorder Buffer (ROB)* à un rythme pouvant atteindre 4 µOPs par cycle. Sur Skylake on a un mécanisme similaire mais, avec 5 µOPs par cycle. Dans Vulcan, Broadcom s'est débarrassé de l'ordonnanceur distribué et l'a remplacé par un ordonnanceur unifié plus efficace, de conception similaire au Skylake. Vulcan utilise un ordonnanceur unifié à 60 entrées contre 97 entrées pour Skylake. Il est intéressant de noter que pour prendre en charge quatre *threads*, Vulcan duplique la plupart de la logique de chaque thread, comme tous les registres, les états architecturaux, les compteurs de programme et les interruptions. Les µOPs prêts de n'importe quel thread sont émis à chaque cycle.

Jusqu'à six µOPs peuvent être envoyés dans les six unités d'exécution de Vulcan à chaque cycle. En ce qui concerne les opérations sur les entiers, jusqu'à trois opérations peuvent être émises par cycle. L'une des **ALU** *(Arithmetic and Logic Unit)* gère également les instructions de branchement. Notez que seule l'ALU du port 1 peut effectuer des opérations sur des entiers complexes (c'est-à-dire la multiplication et la division) en plus des opérations sur des entiers simples. Les deux autres ALU

5. source : https ://en.wikichip.org/wiki/cavium/microarchitectures/vulcan

ne peuvent effectuer que des opérations sur des entiers simples. Vulcan a doublé le nombre d'unités à virgule flottante, qui passe à deux, et les a élargies à 128 bits pour prendre en charge les opérations NEON d'ARM (la conception précédente n'avait qu'une largeur de 64 bits). En théorie, les performances maximales de Vulcan atteignent 8 FLOP(Floating Operation)/cycle ou 8 GFLOPS (Giga Floating Operation per Second) à 1 GHz. Cependant, Skylake promet de meilleures performances avec des unités de 512 bits pouvant produire 32 GFLOP/cycle. Ce qui montre que Skylake peut être jusqu'à quatre fois plus rapide sur des applications exécutant beaucoup de calcul flottant.

Le sous-système de mémoire de Vulcan s'occupe des demandes et de l'ordre de chargement et de stockage. Il y a deux unités de chargement et de stockage, chacune étant capable de déplacer 128 bits de données contre 512 bits sur Skylake grâce à ses unités de 512 bits. L'élargissement des unités a été réalisé afin de prendre en charge plus efficacement des opérations telles que les instructions *Load Pair (LDP)* et *Store Pair (STP)*. Comme sur les anciennes architectures d'Intel, l'opération de stockage est décomposée en deux opérations distinctes : une opération de stockage d'adresse utilisée pour calculer l'adresse effective et enfin l'opération de stockage de données. Vulcan peut émettre un stockage vers l'unité *Store Address* avant que les données ne soient disponibles, ce qui permet de calculer l'adresse et de détecter les conflits d'ordre de mémoire. Une fois que les données sont prêtes, l'opération est réémise à l'unité LSU.

Le Niveau de cache L2 de Vulcan est de 256 KiB, soit le quart de celui de la conception d'Intel Skylake qui est de 1024 KiB, et dispose d'une bande passante L2 vers L1 de 64 octets par cycle dans les deux sens. Il y a un cache L3 de 1 Mio par cœur, organisé en tranches de 2 Mio, pour un total de 32 Mio de cache partagé par l'ensemble de la puce. Le L3 est un niveau exclusif, qui se remplit de lignes de cache L2 évincées. Nous notons une conception similaire chez Skylake, sauf que cette fois-ci, on a 1.375 MiB par cœur pour un total de 22 MiB de L3 (avec 16 cœurs) et que tous les niveaux de caches sont exclusifs.

Il y a 16 tuiles de L3 qui sont dépourvues de lignes de cache sur Vulcan. Bien qu'elles soient divisées en tranches, il n'y a pas de notion d'affinité du cache L3 avec les cœurs. Lors d'un *miss L2 (donnée non trouvée dans le L2)*, un *hash* est utilisé pour déterminer le *hit* (donnée trouvée) de cache L3 d'origine. Une vérification est effectuée pour déterminer si la ligne de cache est trouvée dans cette tuile L3 et si elle est trouvée, elle retourne les données. Cavium a implémenté une version améliorée du protocole MOESI. S'il n'est pas trouvé, un filtre snoop indique la présence des données dans d'autres cœurs. Si elles sont présentes suite à un snoop, le propriétaire transfère la ligne. Lors d'un snoop négatif pour tous les cœurs, une requête DRAM est émise vers le contrôleur mémoire. Cependant, nous n'avons pas plus de détails sur cette implémentation afin de le comparer à celle d'Intel.

Pour conclure cette section, nous dirons que l'étude détaillée des deux architectures nous a permis de voir le rapprochement des deux types de processeurs les plus populaires à savoir x86 et ARM. Donc, en réalité ces deux processeurs sont très similaires dans leurs conceptions à quelques différences près. Cette présentation est indispensable à comprendre pour la caractérisation et une bonne analyse de performance des applications.

1.1.2 Système calculatoire des processeurs modernes

L'objectif de cette section est de présenter une vue générale des éléments et concepts importants pour notre travail de mémoire. Nous présentons un aperçu des éléments ainsi que des techniques d'optimisations matérielles que l'on rencontre sur la microarchitecture des processeurs modernes.

Processeurs multicœur

La vitesse de calcul des processeurs est liée à sa fréquence. Cette dernière a largement contribué à l'évolution de leur performance. Cependant, certaines limites physiques empêchent l'augmentation infinie des fréquences. Il a donc fallu trouver d'autres moyens d'améliorer la performance des processeurs, sans pouvoir accélérer leur fréquence. L'apparition des processeurs multicœurs est une réponse à ce défi.

Le terme **processeur multicœur** est employé pour désigner tout processeur possédant entre deux et quelques dizaines de cœurs (on parle de processeurs *manycore* au-delà). Les différents cœurs sont disposés sur la même puce d'où l'autre appellation utilisée pour désigner ces processeurs de **Chip Multiprocessor (CMP)**.

L'avantage principal de dupliquer un cœur plutôt que de doubler la fréquence d'un seul cœur est la réduction de la consommation électrique et donc plus précisément, la puissance dégagée par effet Joule. Comme pour le pipeline, le gain de performances apporté par l'ajout de cœur s'appuie sur l'amélioration du niveau de parallélisme d'instructions (*ILP*). La difficulté d'utilisation de processeurs multicœurs vient des programmes qui ne sont pas capables par nature d'utiliser ce niveau de parallélisme. Ils doivent donc être programmés pour pouvoir en profiter. Les processeurs *TX2* sont des *manycore* avec 32 cœurs sur un *CMP*. Et il est possible de connecter deux **CPU(Central Processing Unit)** *TX2* de 32 cœurs chacun dans un même serveur. Ce qui fait un total de 64 cœurs disponibles pour un serveur. Chaque cœur possède ses caches L1 et L2 privées et partage le cache L3 avec les autres cœurs.

Multi-threading

Pour économiser davantage les ressources matérielles, on peut étendre le partage des fonctions à des niveaux plus près du processeur. C'est ce qu'on appelle le **SMT (Simultaneous Multi-Threading)** ou **hyper-threading** : les *threads* matériels sont des unités de traitement situées sur le même cœur. Mais toutes les unités fonctionnelles ne sont pas dupliquées. Seules certaines unités à l'instar des registres (registres généraux et registres spéciaux) sont dupliquées. Mais le *pipeline, l'ALU, et autres unités d'exécutions* sont partagés.

Les *threads* matériels peuvent exécuter des flux d'instructions de façon autonome. Par conséquent, le SMT n'améliore pas les performances maximales du matériel puisque différents *threads* partageant les mêmes unités fonctionnelles ne peuvent pas effectuer certaines opérations en même temps. Néanmoins, il peut améliorer l'utilisation du *pipeline* en remplissant les bulles insérées dans le *pipeline* par l'un des *threads* avec des instructions de l'autre *thread*. Le **multithreading** simultané augmente seulement les performances durables du matériel. Des ordonnanceurs matériels sont implémentés sur les architectures modernes afin de mieux équilibrer la charge sur le système.

Sur *TX2*, nous avons un *SMT 4-way*. Ce qui veut dire que sur chaque cœur du *CPU*, nous pouvons activer jusqu'à quatre *threads* (processeurs logiques) au même moment. Ce qui fait qu'on a 256 *threads* pour un serveur en dual *CMP*. Cela peut être très intéressant pour l'exécution des applications parallèles.

1.1.3 Système mémoire des processeurs modernes

La hiérarchie des mémoires est un élément essentiel de l'architecture des ordinateurs, notamment dans le contexte du calcul intensif. En effet, il est inutile d'améliorer les performances du processeur si les performances de la mémoire ne sont pas augmentées en même temps : la vitesse de calcul d'un processeur n'a aucune importance s'il doit constamment attendre la mémoire. Le **mur de la mémoire** est un concept qui explique pourquoi le système mémoire est si critique pour les performances d'un ordinateur.

Problème du mur de la mémoire

L'écart croissant entre les performances de la mémoire et celles des processeurs est souvent appelé *mur de la mémoire (memory wall)*. Deux principales raisons peuvent expliquer l'apparition de cet écart de performance : *l'économie du marché des mémoires et l'évolution des technologies mémoires* [Pourroy, 2020]. Le prix par *gigaoctet* diminue avec la densité des mémoires. Les constructeurs souhaitent avoir des mémoires de plus grande capacité dans des espaces restreints avec des contraintes énergétiques. La **SRAM (Static Random Access Memory)**, moins dense et plus consommatrice est ainsi passée en second plan. Le gain de transistors assuré par la loi de *Moore* a été utilisé pour construire des mémoires plus denses et donc de plus grande capacité. Cependant, l'industrie des microprocesseurs a profité de ces transistors pour construire des processeurs plus performants tandis que l'industrie des mémoires en a profité pour créer des mémoires de plus grande capacité. La deuxième raison provient de l'architecture de *Von Neumann* utilisée dans les architectures modernes. En effet, cette architecture implémente une microarchitecture qui sépare la partie exécution de la partie mémoire. Le processeur a besoin des données provenant de la mémoire pour effectuer les calculs. Conséquence : le bus mémoire devient un *goulot d'étranglement (bottleneck)* pour les performances.

Hiérarchie mémoire

L'objectif des constructeurs est de proposer une mémoire de grande capacité, très rapide à un prix le plus faible possible, ce qui n'est malheureusement le cas d'aucune des technologies existantes. Pour atteindre cet objectif, la solution imaginée par les architectes est l'utilisation de différentes tailles et types de mémoires constituants une hiérarchie de mémoires. La solution est de placer au plus proche du processeur, des mémoires très rapides pouvant répondre instantanément aux accès mémoire. Plus on s'éloigne des unités de calcul, plus la latence d'accès aux modules mémoires augmente, mais plus leur prix diminue rendant possible l'utilisation de module de plus grande capacité. Lorsque le processeur souhaite accéder à une donnée, il vérifie qu'elle se trouve dans son premier niveau de mémoire et, si ce n'est pas le cas, remonte la hiérarchie jusqu'à la trouver. L'apport d'un gain de performances de la hiérarchie réside dans le fait que la donnée accédée doit se trouver le plus souvent possible dans les zones mémoires les plus proches du processeur.

Mémoires Caches

Les caches sont des modules de mémoire généralement en *SRAM* à très faible latence d'accès (quelques cycles). Ces mémoires étant très chères, elles ne peuvent pas être construites en grande quantité. La performance d'une application dépendra fortement de la présence ou non des données nécessaires dans ces mémoires. C'est le concept de *localité*.

Pour allier les avantages et contourner les inconvénients, les processeurs utilisent non pas un, mais plusieurs niveaux de caches de tailles différentes. Le premier niveau de cache est généralement séparé en deux zones mémoire : l'une contenant les instructions et l'autre les données. C'est le seul niveau de la hiérarchie qui stocke différemment les données et les instructions. Sur les processeurs récents, le premier et le deuxième niveau de cache sont privés à chaque cœur. Un troisième et parfois un quatrième niveau de cache sont partagés entre les différents cœurs du processeur. Le partage d'un ou plusieurs niveaux de caches entre différents cœurs a certains bénéfices en programmation parallèle. La communication entre les cœurs est plus rapide, ainsi que la migration d'un *thread* entre deux cœurs partageant un même niveau de cache. Cependant, cela introduit de la complexité pour la *cohérence des caches*. On peut voir sur la figure 1.1 les différents caches présents sur la microarchitecture *Vulcan* de *TX2*. Plus de détails ont été donnés en fin de section 1.1.1 sur le fonctionnement et la gestion des lignes de caches sur les deux microarchitectures Vulcan et Skylake. Cependant, d'autres détails plus poussés sur les algorithmes de gestion des caches *(politique de*

remplacement des lignes de caches, implémentation du filtre snoop, algorithme de recherche de ligne de cache, etc.) ne sont pas disponible dans les documentations que nous avons trouvés.

1.2 Caractérisation des architectures

Cette section présente les méthodes et les outils existants permettant de caractériser le matériel. Dans ce travail, nous nous sommes principalement concentré sur la caractérisation de la microarchitecture des processeurs et du système mémoire. Il existe deux méthodes pour caractériser une architecture : *la méthode théorique et la méthode pratique (benchmarks)* [Pourroy, 2020]. Cette caractérisation peut nous servir à construire des modèles de performance pour l'architecture.

1.2.1 Caractérisation théorique

Cette méthode consiste à utiliser les données communiquées par le constructeur d'une plateforme pour calculer la performance du matériel. On a juste besoin de la documentation de l'architecture afin de calculer les caractéristiques. Cette approche est intéressante, car elle permet de caractériser une architecture sans y avoir accès. Cependant, certaines données peuvent ne pas être disponibles et nécessitent d'avoir accès aux architectures pour les obtenir.

Dans la littérature, il existe des outils disponibles sur les distributions Linux *(lscpu , cpumap , cpuid, hwloc, numactl)* qui peuvent alors être utilisés pour obtenir plus d'informations sur l'architecture, notamment lorsque l'on n'a pas accès à la documentation. Néanmoins, ces outils peuvent ne pas supporter toutes les architectures et certaines données peuvent toujours manquer.

1.2.2 Caractérisation par Benchmarks

Dans le domaine informatique, un **benchmark** est un code, ou un ensemble de codes, permettant de mesurer la performance d'une solution et d'en vérifier ses fonctionnalités [Pourroy, 2020]. Ces codes sont en majorité libres de droits et sont généralement utilisés pour comparer les performances des plateformes.

Cette seconde méthode nécessite d'avoir accès à l'architecture dans le but d'y exécuter des codes (*benchmarks*). Contrairement à la première méthode (caractérisation théorique), l'utilisation de ces applicatifs permet de mesurer une performance réellement atteignable sur l'architecture. En effet, à cause de la complexification des microarchitectures, il est très rare que la performance mesurée soit égale à la performance théorique calculée à partir des données techniques de l'architecture. Dans cette partie, nous discutons de quelques *benchmarks* les plus populaires de la littérature.

— **HPL (High Performance Linpack)** [Dongarra et al., 2003] C'est un des *micro benchmarks* les plus utilisés (par exemple pour construire le classement du *Top500*[6]). Il est utilisé pour mesurer le nombre maximum de *FLOPS* qu'un processeur est capable de fournir pour la résolution d'un système linéaire d'équations. Bien que mondialement utilisé, le *HPL* a un principal défaut qui est d'estimer la performance d'une plateforme en ne mesurant que sa capacité de calcul. Cependant, la performance de la majorité des applications est limitée par la performance de la bande passante. La mesure du HPL n'est donc pas la plus représentative de la performance réellement atteignable des architectures.

— Pour pallier les faiblesses de HPL, [Report et al., 2013] proposent un nouveau benchmark nommé **HPCG (High Performance Conjugate Gradient)**. *HPCG* permet de couvrir de

6. Classement du Top500 - https ://www.top500.org/

nombreux motifs de communication (globale et voisinage) et de calculs (mise à jour de vecteur, multiplication de matrices creuses). L'objectif principal était alors de pouvoir produire, grâce à ce nouveau *benchmark*, un classement des supercalculateurs représentatif de leur performance pour exécuter des applications réelles. Le classement du *Top500* est aujourd'hui publié avec les valeurs obtenues par *HPL* et par *HPCG*. L'avantage de ce *benchmark* pour nos travaux est la prise en compte du noyau de calcul **SpMV (Sparse Matrix Vector Multiplication)** qui possède des schémas d'accès à la mémoire irréguliers et que l'on retrouve dans la majorité des applications HPC. Il est donc plus sensible à la bande passante mémoire.

— **SPEC CPU2017** [Bucek et al., 2018] est une nouvelle génération de benchmarks produite par SPEC (*Standard Performance Evaluation Corporation*). L'objectif principal de ces codes était de caractériser les performances du processeur, de la hiérarchie mémoire et du compilateur. La version 2017 possède 43 benchmarks qui sont portés sur plusieurs architectures dont AMD64, Intel IA32, Power ISA ou SPARC. Les différents benchmarks ont un domaine d'application spécifique (compression vidéo, rendu 3D...). Malheureusement, le domaine des applications de recommandation n'est pas présent pour ces benchmarks et ils sont payants.

— **STREAM** [McCalpin, 1995] Le benchmark STREAM est sûrement un des benchmarks les plus connus et les plus utilisés dans le domaine du HPC. Le code STREAM permet de mesurer la bande passante mémoire atteignable grâce à l'implémentation de quatre noyaux : COPY (c = a), SCALE ($b = \alpha \times c$), ADD (c = a+b) et TRIAD ($a = b + \alpha \times c$). Les résultats sont donnés en GB/s et contiennent à la fois les opérations de lecture et d'écriture. Il est généralement accepté que la mesure donnée pour l'opération de triad correspond à la bande passante maximale atteignable par l'architecture [Pourroy, 2020]. Cependant, les applications réelles utilisant des motifs d'accès bien plus complexes, cette mesure n'est pas représentative de la performance réellement atteignable des architectures.

Benchmark	**avantages**	**Limites**
HPL	simple avec un seul résultat	ne mesure que la capacité de calcul
HPCG	prise en compte des accès irréguliers	beaucoup de résultats difficiles à synthétiser
SPEC CPU2017	*micro-benchmarks* répartis dans plusieurs domaines	payant et pas disponible pour ARM TX2
STREAM	Simple à comprendre et à exécuter ; se concentre sur la mémoire	ne prend en compte que les motifs d'accès mémoire réguliers

TABLE 1.1 – Récapitulatif des *benchmarks* de la littérature avec les principaux avantages et limites de chaque benchmark.

Dans la suite, nous allons utiliser le *benchmark* **STREAM** pour notre modélisation bien que ses motifs d'accès ne soient pas représentatifs de ceux des algorithmes étudiés. Mais cela nous permettra d'être plus optimiste et d'avoir des valeurs un peu plus représentatives de la performance maximale possible sur l'architecture.

1.3 Analyse de performance des applications

Afin de permettre l'analyse et l'optimisation des codes, deux étapes principales sont étudiées : identifier les **hot spots** des applications et identifier les **goulots d'étranglement** de l'architecture.

C'est l'optimisation de ces **hot spots** de codes qui aura le plus d'impact sur l'amélioration de la performance de l'application. D'où la nécessité de pouvoir identifier et caractériser ces zones de code. Il est important d'identifier les goulots d'étranglement afin de mesurer l'écart entre la performance réelle de l'application et la performance prédite par des modèles établis ou encore la performance théorique atteignable par l'architecture.

Pour obtenir les informations nécessaires à l'analyse de la performance d'une application, deux approches peuvent être employées : **l'analyse statique et l'analyse dynamique**.

1.3.1 Analyse statique

Dans ce type d'analyse, on n'a pas besoin d'exécuter une application ou d'accéder à la microarchitecture pour obtenir les informations. Ce type d'analyse peut permettre de détecter des bogues ou de modéliser la performance de l'application grâce à l'utilisation de modèles analytiques. L'analyse peut être réalisée à partir du code source, ou des instructions assembleur généré par le compilateur. Il existe des outils d'analyse statique afin de faciliter cette tâche :

— **MAQAO (Modular Assembly Quality Analyzer and Optimizer)** [Djoudi et al., 2005] est un outil qui permet de détecter les boucles responsables des *hot spots* et d'en analyser leur performance. MAQAO comprend le développement de l'outil **CQA** [Charif-Rubial et al., 2014] permettant d'évaluer la qualité du code assembleur et de projeter la performance maximale pour une architecture donnée. Il est ainsi possible de connaître les opportunités de vectorisation et des gains potentiels pour les *hot spots* de l'application. Cependant, cet outil présente plusieurs limites à savoir : l'absence de documentation pour pouvoir l'utiliser, la dépendance aux bibliothèques qui ne sont pas disponibles sur certaines architectures, etc. Cela rend son utilisation impossible pour certaines plateformes, notamment sur TX2.

— **IACA(Intel Architecture Code Analyzer)** [Israel and Gideon,] permet de réaliser une analyse statique d'un code grâce à l'ajout de marqueurs dans le code source. Il permet, pour un noyau identifié, de détecter la présence des dépendances entre plusieurs itérations de boucle et de donner une estimation des performances (débit d'instructions, saturation des ports de l'ALU, ...). Cependant, il est nécessaire d'avoir identifié les zones de hot spots pour les annoter, et il ne fonctionne que pour des architectures Intel. Le projet a été abandonné en 2019, donnant lieu au développement de **llvm-mca** [LLVM,] qui propose la possibilité d'utiliser les compilateurs **clang** et **GNU GCC**. En plus, **llvm-mca** est disponible pour les architectures *Intels* et *ARM* dont notamment *TX2*

1.3.2 Analyse dynamique

Par contre, dans ce type d'analyse, il est nécessaire d'exécuter une application afin d'accéder à la microarchitecture pour obtenir plus d'informations qu'il n'est possible d'avoir avec l'analyse statique. Les informations peuvent être obtenues grâce à l'implémentation matérielle de registres permettant de suivre l'activité de la microarchitecture. C'est sur ce type d'analyse que se porte notre choix afin d'avoir plus de précisions dans les résultats. Plusieurs outils existent dans la littérature pour faciliter l'analyse :

— [Pourroy, 2020] propose **Oprofile++**, un outil de suivi de performance basé sur l'outil **Oprofile** [Levon and Elie, 2004] intégré dans les systèmes Linux. Celui-ci fonctionne en deux étapes. D'abord suivre les performances du processeur lors de l'exécution d'une application en utilisant les compteurs matériels programmés pour compter le nombre de cycles et le nombre d'instructions exécutées. Ensuite extraire les *hots spot (noyaux d'exécutions)* de l'application. Une fois ces zones de code identifiées, l'outil fait correspondre les événements enregistrés lors de l'exécution avec les instructions assembleurs qui en sont responsables. Il est

ensuite possible de quantifier des opportunités d'amélioration (optimisation du code), mais aussi de prédire la performance de ces boucles en fonction d'une amélioration du matériel ou du logiciel. Cependant, cet outil est limité par la dépendance aux compteurs matériels (qui peuvent être différents sur d'autres architectures) ainsi que de *Oprofile* (dont les nouvelles versions ne sont pas compatibles pour l'exécution de *Oprofile++*). Cela peut donc nécessiter une adaptation afin d'être utilisé sur les nouveaux noyaux Linux.

— [Pourroy, 2020] propose également **YAMB**, un outil de suivi de performance basé sur l'outil **perf events** [Weaver, 2013] des systèmes Linux. YAMB mesure l'activité des différents contrôleurs mémoire (en lecture et en écriture) ainsi que le nombre d'événements *miss (la donnée recherchée dans le cache n'a pas été trouvée)* dans le dernier niveau de cache (LLC). L'évolution du trafic mémoire peut ensuite être affichée sous forme de graphique. L'outil propose l'utilisation d'une bibliothèque permettant d'annoter facilement certaines zones du code pour faciliter la lecture du graphique [Pourroy, 2020]. Cependant, l'une des limites de YAMB est la dépendance aux compteurs matériels d'Intel qu'on ne retrouve pas sur toutes les architectures. YAMB peut donc nécessiter des adaptations pour pouvoir être utilisé sur une nouvelle architecture. Et malheureusement sur les architectures TX2, nous n'avons pas trouvé de compteur pour la mesure des défauts de caches de niveaux L3.

— [Treibig et al., 2010] propose **LIKWID**, un ensemble de modules utilisables en ligne de commande pour aider les développeurs dans le travail de caractérisation et d'analyse de performances. Les modules fonctionnent sur la majorité des distributions Linux grâce à sa faible dépendance à des librairies externes : il ne nécessite que le compilateur *GNU compiler*. L'outil possède 11 modules pouvant être groupés en trois catégories : analyse de performance, caractérisation de plateforme et contrôle de l'exécution. Un de ces outils est important pour analyser les performances d'une application : *likwid-perfctr*. L'avantage de cet outil est de pouvoir utiliser des groupes d'événements déjà existants comme par exemple le groupe **"MEM"** [7] qui permet de donner des informations sur le trafic mémoire en termes de bande passante mémoire et du volume de données. L'outil peut être utilisé pour mesurer la performance d'une application particulière utilisant la programmation multicœur. *Likwid* propose une API permettant d'annoter le code pour ne mesurer la performance que de certaines régions de l'application.

Outil	**avantages**	**Limites**
Oprofile++	extraction des points chauds	forte dépendance à *oprofile* qui évolue avec le noyau linux ; ne tient pas compte de la mémoire
YAMB	Résultats graphique facile à visualiser	forte dépendance aux compteurs matériels d'Intel qui ne sont pas malheureusement présents sur ARM TX2
LIKWID	fournis plusieurs résultats et assez complet ; mise à jour régulière	/

TABLE 1.2 – Récapitulatif des outils d'analyse de la littérature avec les principaux avantages et limites de chaque outil.

7. https ://github.com/RRZE-HPC/likwid/blob/master/groups/arm8_tx2/MEM.txt

Dans ce mémoire, nous travaillons essentiellement avec LIKWID pour la mesure des performances réelles, afin de le comparer aux modèles prédictifs établis.

Dans ce chapitre, nous avons fait un tour d'horizon des éléments et notions indispensables à la compréhension de notre travail. Nous avons également fait un état de la littérature sur les méthodes et outils de caractérisation d'architectures et d'analyse de performances qui permettent de comprendre le cadre de nos travaux et de nous positionner par rapport aux travaux existants. Cette caractérisation et analyse nous permettent de construire des modèles de performance pour la prédiction ou comparaison sur les architectures. Ainsi, pour le chapitre suivant nous allons continuer avec un état de l'art des modèles prédictifs qui se rapprochent le plus de notre problème de recherche qui est celui d'établir des modèles prédictifs pour analyser les performances des applications de recommandation.

Chapitre 2: État de l'art sur les modèles de performance des architectures

Dans ce chapitre, nous présentons quelques modèles de performances qui existent dans la littérature afin de modéliser les performances des applications sur une architecture.

Notre étude est basée sur des approches analytiques et elles sont assez traditionnelles pour construire des modèles de performance. Elles exigent une compréhension profonde des algorithmes qui sont utilisés, des détails sur leur mise en œuvre et une bonne connaissance de la structure dynamique de l'exécution de l'application. Ces méthodes sont plus utilisées dans le domaine de la recherche que dans le milieu industriel. Toutefois, elles continueront à être utilisées jusqu'à ce que d'autres méthodes automatisées deviennent suffisamment matures pour atteindre le niveau désiré de précision. Ces modèles sont généralement utilisés pour prédire la performance d'une application sur une architecture donnée. Sur une plus petite échelle, ces modèles peuvent également être utilisés pour valider des modèles construits avec d'autres techniques analytiques ou non.

2.1 Modèle du Roofline

Les applications du HPC sont le plus souvent limitées par la mémoire (**MEMORY BOUND**) ou la vitesse de calcul (**COMPUTE BOUND**). Les applications limitées par la mémoire repoussent les limites de la bande passante du système, tandis que les applications limitées par la vitesse de calcul repoussent les capacités de calcul du processeur. Pour modéliser cette limite de performance pour les applications, [Williams et al., 2009] proposent le modèle du **Roofline**. Le modèle donne la bande passante maximale réalisable pour la hiérarchie mémoire sur une machine en évaluant l'*OI(Operational Intensity)* et un nombre maximal de *FLOPS* pour l'architecture. Le modèle Roofline repose sur l'hypothèse fondamentale que le transfert de données à travers la hiérarchie mémoire se chevauche parfaitement avec l'exécution *in-core*. La figure 2.1 montre une représentation graphique du modèle du roofline.

La force de cette approche est de montrer rapidement au programmeur si son application est efficace ou non. Dans le cas échéant, il sait s'il doit travailler sur l'optimisation des instructions de calcul ou des accès mémoire. Bien qu'ayant reçu de nombreuses améliorations qui permettent de prendre en compte les informations de la hiérarchie de caches, ce modèle doit être utilisé pour commencer l'analyse de performance. Cependant, il ne permet pas de modéliser ni de comprendre finement la raison d'une performance. La majorité des applications étant limitée par la bande passante mémoire, il est rare d'utiliser ce modèle pour modéliser la performance des unités de calculs. Dans le cadre de notre étude, il peut être utile pour confirmer que les applications de recommandation étudiées sont **MEMORY BOUND**.

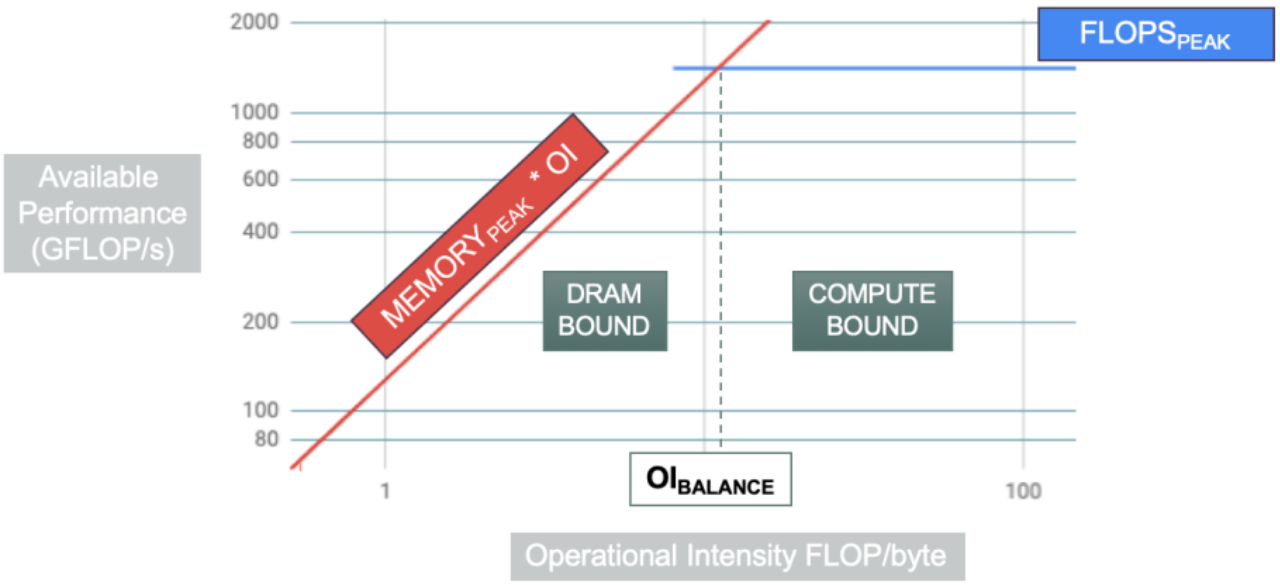

FIGURE 2.1 – Modèle du Roofline pour l'architecture TX2. En fonction de son intensité opérationnelle, la performance d'un code sera limitée par la bande passante (*memory bound*) ou par le processeur (*compute bound*). Source : [Pourroy, 2020]

2.2 Modèle ECM

Le modèle **ECM***(Execution Cache Memory)* [Hager et al., 2016] repose sur la même idée fondamentale que le modèle Roofline, à savoir que le transfert de données ou l'exécution des instructions, selon ce qui prend le plus de temps, détermine le temps d'exécution de la boucle. Cependant, contrairement au modèle Roofline, il abandonne l'hypothèse d'un goulot d'étranglement unique. Les transferts de données à travers la hiérarchie de la mémoire sont séquentiels dans les niveaux de mémoire d'un cœur et contribuent donc à la réduction de la performance totale en se chevauchant les uns des autres. Ce modèle présente également une métrique plus précise que les *FLOPS* : **le nombre de cycles par ligne de cache (cy/CL)**, l'unité de mesure qui donne plus d'importance à la hiérarchie de la mémoire.

Le modèle ECM prend en compte l'analyse de la condition des couches (LC) qui permet de prédire les besoins en cache pour les codes **stencil**. La base de l'analyse LC est la politique de remplacement du cache par le moins récemment utilisé (LRU), qui n'est pas parfaitement mise en œuvre dans les caches réels et de grandes tailles. En prenant en compte les décalages relatifs aux accès mémoire et en supposant des incrémentations séquentielles pendant les itérations ultérieures, les accès **hit/miss** peuvent être prédits en fonction des tailles des caches. Le calculateur LC peut prédire les zones de tuiles optimales pour le blocage spatial des codes stencil.

Cependant, il ne peut ni prédire la taille optimale de la tuile le long de chaque dimension, ni être étendu aux stratégies de blocage temporel. Ainsi, la principale limite de ce modèle est de ne pas pouvoir modéliser la localité temporelle des données. Une autre limite que l'on peut noter est le mécanisme de gestion de cache (caches *Victimes* sur des architectures récentes par exemple) qui n'est pas prise en compte.

2.3 Modèle SMM

[Pourroy, 2020] a développé un modèle de performance simple, permettant de modéliser et de valider les performances d'un code facilement : le **SMM** *(Simple Memory Model)*. Pour réaliser cette modélisation, le développeur doit avoir accès au code source de l'application à porter. Pour un noyau de calcul donné, il faut compter le nombre d'accès mémoire en distinguant les accès en lecture et ceux en écriture. Il est conseillé d'appliquer ce modèle sur les noyaux de calcul des applications car, l'appliquer sur la totalité du code serait très long. Ce modèle utilise deux paramètres (la taille des données **DATAsize** et la bande passante mémoire maximale **MEMORYpeak**) afin de calculer le temps *théorique optimal* pour l'exécution d'une application. Ainsi, on a :

$$TEMPSoptimal = \frac{DATAsize}{MEMORYpeak}$$

Le modèle suppose que le code utilise un algorithme parfait (utilisation parfaite de la localité des données), que la compilation du code a été réalisée avec un compilateur parfait et qu'il est exécuté sur une plateforme parfaite [Pourroy, 2020]. La première limite de ce modèle vient de ces suppositions, car lorsqu'un défaut apparaît à un des niveaux énumérés précédemment, la performance s'éloigne radicalement de la performance optimale. Et de nos jours, il est très difficile que ces suppositions soient respectées. La deuxième limite de ce modèle ne tient pas compte du parallélisme avec plusieurs cœurs et n'est donc adapté qu'à des programmes séquentiels.

2.4 Loi d'Amdahl

Le premier modèle couramment utilisé pour déterminer le gain de performance des programmes parallèles a été conçu en 1967 par Amdahl [Amdahl, 1967] et porte le nom de **Loi d'Amdahl**. Cette loi est utilisée pour calculer l'accélération *(speedup)* théorique maximale possible S d'une version parallèle d'un programme comparé à sa version séquentielle, en utilisant la formule :

$$S = (\frac{1}{r_s + \frac{r_p}{n}})$$

où r_s est le pourcentage de la partie séquentielle du programme, r_p correspond à celui de la partie parallèle du programme et n est le nombre de processeurs utilisé dans la version parallèle.

Cette loi est souvent utilisée en calcul parallèle pour prédire l'accélération théorique lors de l'utilisation de plusieurs processeurs. Par exemple, si un programme a besoin de 20 heures d'exécution sur un processeur mono-cœur et qu'une partie du programme qui requiert une heure d'exécution ne peut pas être parallélisée, même si les 19 heures ($r_p = 95\%$) d'exécution restantes peuvent être parallélisées, quel que soit le nombre de processeurs utilisés pour l'exécution parallèle du programme, le temps d'exécution minimal ne pourra passer sous cette heure critique. Ainsi, l'accélération théorique est limitée au plus à 20 (1/(1 - p) = 20) [1].

La partie séquentielle d'un code représentant le temps d'exécution associé à toutes les parties du code qui ne peuvent être exécutées uniquement que sur un seul processeur. La loi d'Amdhal ne tenant pas compte du coût des transferts des messages et d'autres frais de parallélisations, il est surtout utilisé pour évaluer les performances d'un programme dans le cadre de la programmation à mémoire partagée. Ainsi, nous l'utiliserons dans notre méthodologie d'analyse pour prédire l'accélération de l'exécution en fonction de l'évolution du nombre de *threads* sous une parallélisation OpenMP.

1. source : https ://fr.wikipedia.org/wiki/Loi_d%27Amdahl

Récapitulatif des modèles prédictifs.

Le tableau 2.1 nous donne un récapitulatif des principaux modèles analytiques et loi en termes d'avantages et limites.

Modèle	avantages	Limites
Roofline	Permet de savoir rapidement si le code est efficace ou pas	Ne permet pas de comprendre les raisons de la mauvaise performance du code
ECM	Modélise tous les niveaux de la hiérarchie	Adapté aux motifs d'accès régulier des codes *stencil* ; Le transfert des données dans la hiérarchie cause un problème pour les caches victimes.
SMM	Facile à établir et prise en compte du volume de trafic de données	Trop de suppositions qui peuvent rendre les prédictions très éloignées de la réalité
Amdahl	Permet d'analyser le parallélisme	Ne prend pas en compte les coûts de parallélisation

TABLE 2.1 – Récapitulatif des modèles analytiques de la littérature avec les principaux avantages et limites de chaque modèle.

le modèle ECM est difficile à mettre en œuvre sur nos architectures de test à cause du fait que le schéma de circulation des données dépend entièrement de la gestion du cache. Dans le modèle ECM original, les données vont de la mémoire DRAM vers le cache L3, puis au L2 et ensuite au L1. Mais avec nos deux architectures, seul le L2 alimente le L3. Pour éviter les résultats erronés, nous avons laissé ce modèle de côté pour la suite. Ainsi, nous allons utiliser les modèles SMM et Amdahl dans la suite pour la prédiction des performances grâce à une méthodologie en plusieurs étapes que nous avons élaborée.

Chapitre 3: Modélisation des performances des applications de recommandation

3.1 Principe des Applications de recommandation

Depuis quelques décennies avec la croissance rapide d'internet, de nombreuses plateformes en ligne proposent de grands nombres de produits à leurs utilisateurs. Ces entreprises opèrent dans des domaines très variés et sont dédiées à des activités diverses comme la vente de produits physiques et/ou numériques, l'écoute des chansons et vidéos à la demande, et autres. Le nombre de produits sans cesse grandissant implique que, pour un utilisateur, la sélection d'un produit qui l'intéresse est un problème réel. Pour y remédier, les systèmes de recommandation sont devenus l'une des solutions les plus utilisées, car ils proposent à chaque utilisateur un petit ensemble de produits qu'il est le plus susceptible de consommer dans un avenir proche. Ceci permet de réduire le temps de recherche des utilisateurs et facilite leurs orientations dans cette masse de données.

3.1.1 Approches de constructions des systèmes de recommandation

Plusieurs approches existent dans la littérature pour construire de tels systèmes. Nous avons les approches basées sur le filtrage collaboratif, les approches basées sur le contenu et les approches hybrides. Mais l'approche qui nous intéresse et qui est la plus utilisée est le filtrage collaboratif. Dans cette approche, on a plusieurs techniques de construction à savoir : les techniques basées sur la mémoire, sur l'apprentissage automatique et sur les graphes de recommandation [Nzekon Nzeko'o, 2019]. Dans le cadre de notre mémoire, nous nous concentrons sur l'étude des performances de cette technique basée sur les graphes de recommandation.

3.1.2 Approche basée sur les graphes de recommandation

Les graphes sont très utilisés pour étudier la structure et la composition de divers systèmes. Un graphe est constitué d'un ensemble d'éléments appelés nœuds ou sommets et de connexions appelées arêtes ou liens. La recherche des associations transitives dans le contexte des systèmes de recommandation est généralement mise en œuvre à l'aide d'un modèle basé sur des graphes pour deux raisons : un graphe est facilement interprété et fournit un cadre plus naturel et intuitif pour différents types d'applications. Aussi, de nombreux algorithmes basés sur des graphes peuvent être directement implémentés dans différents domaines comme le e-commerce ou les réseaux sociaux.

Les graphes de recommandation définis pour le calcul des recommandations *top-N (i.e recommander N meilleurs produits à un utilisateur selon la métrique choisie)* à partir des données implicites ont les mêmes étapes que les autres systèmes de recommandation basés sur la mémoire comme *KNN* : une étape pour établir les corrélations entre les entités et une seconde étape pour le calcul des recommandations. Dans ce contexte, la première étape est celle de la représentation des données par un graphe et la seconde étape correspond à l'application d'un algorithme de calcul des recommandations à partir du graphe construit précédemment [Nzekon Nzeko'o, 2019].

Construction du graphe

Le principal graphe de recommandation qui est construit à partir de la matrice des notes binaires est le graphe biparti classique. Dans ce graphe, chaque utilisateur (respectivement chaque produit) est représenté par un nœud. Lorsque $R_{u,i} = 1$, le nœud de l'utilisateur u est relié au nœud du produit i par une arête (u, i). Ce graphe est fréquent comme modèle de représentation des données dans plusieurs travaux sur les données implicites.

Une fois les graphes de recommandation construits, on s'en sert pour déterminer les recommandations. Pour chaque type de graphe utilisé, il existe divers algorithmes de calcul des recommandations *top-N*.

Calcul des recommandations

Lorsque le graphe de recommandation est le graphe biparti, l'hypothèse de recommandation repose sur la proximité de l'utilisateur cible aux prochains produits qu'il va sélectionner. Ainsi, l'objectif est de recommander les N produits que l'utilisateur cible n'a pas encore sélectionnés et qui sont les plus proches de lui dans le graphe de recommandation. Suivant ce principe, la plupart des algorithmes de recommandation sur les graphes bipartis sont basés sur la marche aléatoire dans laquelle le nœud source est l'utilisateur ciblé.

Les variantes de la marche aléatoire reposent sur la différence dans la diffusion des poids d'un nœud à l'autre et sur le nombre de pas à effectuer. On a par exemple l'algorithme **Injected Preference Fusion (IPF)** dans lequel les poids sont calibrés dans la propagation et le nombre de pas limité à 3 ou le **PageRank** [Page et al., 1999] initial dans lequel le nombre de *pas* dépend d'un seuil de convergence. D'autres algorithmes comme Katz et **HITS (Hyperlink-Induced Topic Search)** peuvent également être utilisés [Nzekon Nzeko'o, 2019].

Pour l'application de recommandation que nous avons étudiée, nous avons utilisé le *PageRank* personnalisé pour calculer les recommandations. Nous pouvons noter dans cet algorithme l'utilisation des matrices creuses qui conduit à des motifs d'accès irréguliers à la mémoire.

3.2 Description des applications étudiées

Dans cette section, nous présentons une brève description des applications (GraFC2T2 et PageRank) et algorithmes (PageRank, SpMV et Similarité de Jaccard) étudiés afin de faire des analyses préliminaires, de comprendre de façon générale le cadre de notre travail et faire des implémentations. Pour l'application **GraFC2T2** [1], notre implémentation a été effectué en groupe de 3 avec deux autres camarades. Cependant, notre contribution dans cette collaboration a été le développement du module de calcul du PageRank et celui du calcul de la similarité de Jaccard pour l'enrichissement de la qualité de recommandation. Ainsi, nous ne présentons que ces parties dans ce manuscrit.

3.2.1 Application GraFC2T2

La plupart des systèmes de recommandation basés sur les graphes ignorent les informations sur la confiance, et lorsque ces systèmes considèrent des informations secondaires, ce sont soit des informations sur le contenu, soit la dimension temporelle et rarement les deux simultanément. Ces systèmes sont limités, car l'intérêt qu'un utilisateur a pour un produit peut être dû aux caractéristiques de ce produit, ou peut changer avec le temps et peut également être fortement lié aux opinions de ceux à qui il fait confiance.

1. https ://github.com/nzekonarmel/GraFC2T2

Description de GraFC2T2

Dans cette section, les données considérées contiennent un ensemble U d'utilisateurs, un ensemble I de produits, un ensemble C d'attributs de description des produits et que les actions des utilisateurs sur les produits sont observées durant un intervalle de temps T. De plus, nous avons une fonction qui exprime les relations de confiance entre les utilisateurs de telle sorte que $trust(u, v) \in [0, 1]$ représente le niveau de confiance que l'utilisateur u accorde à l'utilisateur v.

Les actions des utilisateurs sur les produits sont modélisées par un flot de liens L inclus dans $T \times U \times I$ où chaque lien $(t, u, i) \in L$ peut représenter un achat (u a acheté le produit i à l'instant t), un intérêt pour un produit audiovisuel (comme regarder un film ou écouter une chanson), ou une autre action de sélection d'un produit par un utilisateur suivant le contexte d'application.

Graphe biparti classique

Le premier graphe de base considéré est le graphe biparti classique (BIP), c'est un graphe biparti **(U, I, E) où U et I** sont respectivement l'ensemble des utilisateurs et des produits. $E \subseteq U \times I$ est l'ensemble des arcs définis par $E = (u, i), (i, u) : \exists t \in T, (t, u, i) \in L$. En d'autres termes, **u** est lié à **i** dans BIP si l'utilisateur **u** a sélectionné le produit **i** pendant la période d'observation T.

D'autres types de graphes STG et LSG ont également été étudiés mais nous avons utilisé dans ce document le graphe BIP car il est plus simple à comprendre.

Architecture globale de GraFC2T2

Les graphes de recommandation sont construits à partir de trois composants : un graphe de base modélisant les relations utilisateur-produit, des attributs descriptifs des produits et une fonction de pénalisation temporelle des arcs les plus vieux. Ensuite, des recommandations top-N sont calculées à partir du graphe construit et en utilisant le *PageRank personnalisé* et les informations basées sur la confiance [Nzekon Nzeko'o, 2019]. La figure 3.1 illustre le fonctionnement global de ce framework (cadre général d'exécution).

3.2.2 La similarité de Jaccard

L'indice et la distance de Jaccard sont deux métriques utilisées en statistiques pour comparer la similarité et la diversité entre des échantillons. Elles sont nommées en l'honneur du botaniste suisse Paul Jaccard [Jaccard,].

Description formelle

L'indice de Jaccard (ou coefficient de Jaccard) est le rapport entre le cardinal (la taille) de l'intersection des ensembles considérés et le cardinal de l'union des ensembles [Jaccard,]. Il permet d'évaluer la similarité entre les ensembles. Soient deux ensembles $v1$ et $v2$, l'indice de Jaccard de ces deux ensemble est : $J(v1, v2) = \frac{|v1 \cap v2|}{|v1 \cup v2|}$

Similarité entre des ensembles binaires

L'indice de Jaccard est utile pour étudier la similarité entre des objets constitués d'attributs binaires. Cela s'applique dans notre cas, car les utilisateurs sont représentés par des vecteurs binaires avec la valeur 1 indiquant que l'utilisateur ait noté le produit à l'indice associé.
Soient deux séquences $v1$ et $v2$, chacune avec n attributs binaires. Chaque attribut (les produits dans notre cas) peut être à 0 ou 1. On a ainsi : $v1 = (a_1, a_2, ..., a_n)$; $v2 = (b_1, b_2, ..., b_n)$.

On définit plusieurs quantités qui caractérisent les deux ensembles :

— M_{11} représente le nombre d'attributs qui valent 1 dans v1 et 1 dans v2 ;

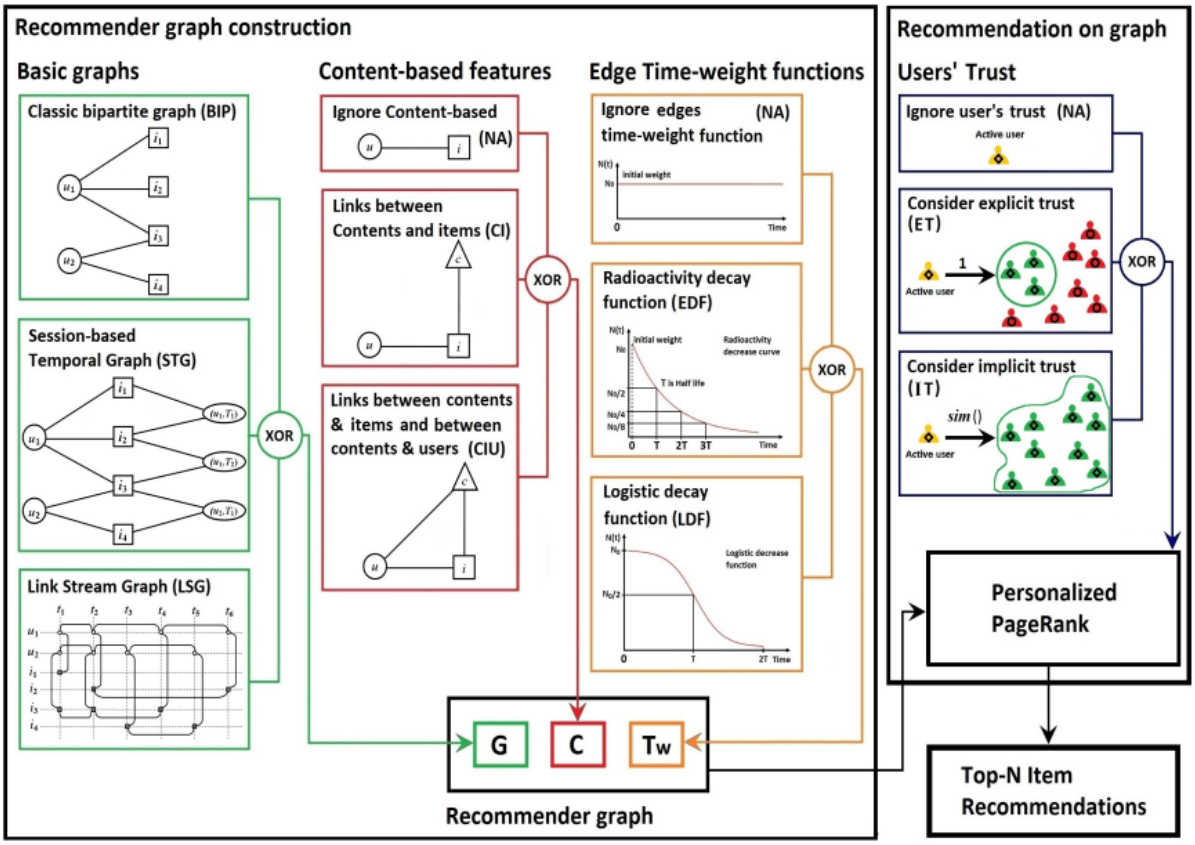

FIGURE 3.1 – Architecture générale du framework GraFC2T2. Source : [Nzekon Nzeko'o, 2019]

— M_{01} représente le nombre d'attributs qui valent 0 dans v1 et 1 dans v2 ;

— M_{10} représente le nombre d'attributs qui valent 1 dans v1 et 0 dans v2 ;

— M_{00} représente le nombre d'attributs qui valent 0 dans v1 et 0 dans v2.

Chaque paire d'attributs doit nécessairement appartenir à l'une des quatre catégories, de telle sorte que : $M_{11} + M_{01} + M_{10} + M_{00} = n$. L'indice de Jaccard devient : $J = \dfrac{M_{11}}{M_{01} + M_{10} + M_{11}}$.

Le code ci-dessous présente notre implémentation naïve séquentielle de cet algorithme dans l'application GraFC2T2 en C++. Dans cette implémentation, on a : $a = M_{11}; b = M_{01} + M_{10}$ Et **J** c'est notre score.

```
// Function to return the Jaccard index
double jaccard_similarity_score(std::vector<double> v1, std::vector<double> v2){
    int size_v1 = v1.size(), a = 0, i = 0, b = 0;
    for(i=0; i<size_v1; i++){
        double v1_i = v1[i], v2_i = v2[i];
        if((v1_i==v2_i) && (v1_i==1.0)){
            a++;
        } else if (v1_i!=v2_i) {
            b++;
        }
    }
    // Calculate the Jaccard index using the formula
    double score = (a + b > 0) ? (double)a / (double)(a + b) : 1.0;
```

```
    return score;
}
```

Le code de l'**Annexe A.1** présente le calcul de la matrice de similarité pour tous les utilisateurs par appel de la fonction ci-dessus.

3.2.3 Application PageRank

Le principe de base est d'attribuer à chaque page une valeur (ou score) proportionnelle au nombre de fois que passerait par cette page un utilisateur parcourant le graphe du Web en cliquant aléatoirement, sur un des liens apparaissant sur chaque page. Ainsi, une page a un PageRank d'autant plus important qu'est grande la somme des PageRank des pages qui pointent vers elle (elle y comprise, s'il y a des liens internes). Le PageRank est une mesure de centralité sur le réseau du web.

Formellement, le déplacement de l'utilisateur est une marche aléatoire sur le graphe du Web, c'est-à-dire le graphe orienté dont les sommets représentent les pages du Web et les arcs les hyperliens.

Nous utilisons l'algorithme **Temporal Personalized Random Walk (TPRW)** [Xiang et al., 2010] pour le calcul des recommandations temporelles (i.e des recommandations avec prise en compte de la dynamique temporelle des utilisateurs). Cet algorithme correspond à l'équation suivante :

$$PR = \alpha \cdot M \cdot PR + (1 - \alpha) \cdot d$$

où PR est le vecteur de *PageRank* ; M est la matrice de transition du graphe ; α est le facteur d'amortissement de la personnalisation ; et d est le vecteur de personnalisation. Pour recommander des produits à un utilisateur u à l'instant t, le vecteur de personnalisation d est configuré en fonction du graphe de base (BIP, STG ou LSG). Une fois que le vecteur d est configuré, le PageRank est exécuté sur le graphe de recommandation construit afin de calculer des recommandations top-N à proposer à u à l'instant t. À la fin de l'exécution du PageRank, l'intérêt que u accorde à un produit i est estimé par la valeur du PageRank du nœud i dans les graphes BIP et STG et par la somme des valeurs du PageRank des nœuds du type produits *(t',i)* associés à i. Ainsi, le système recommande à u les N produits pour lesquels l'intérêt estimé de u est le plus grand. Le code de l'**Annexe A.2** montre l'implémentation de la boucle principale de PageRank dans GraFC2T2.

L'analyse de cet algorithme de pageRank nous a conduit au problème de stockage des matrices creuse pour la matrice de transition M. En plus, la multiplication avec les coefficients du PageRank nous conduit à un problème de multiplication matrice creuse vecteur ($SpMV$). Ainsi, pour la section suivante, nous présentons les deux formats que nous avons implémentés.

3.2.4 La multiplication matrice creuse vecteur (SpMV)

Nous avons restreint la modélisation à l'algorithme du pageRank et pour cela nous avons modifié l'implémentation originelle du **PageRank** [2] afin de pouvoir paralléliser le format CSR et ensuite nous avons implémenté le format COO afin de comparer les performances au CSR et de valider la performance de nos modèles prédictifs. Nous avons choisi ces deux formats, car ce sont les formats classiques, les plus populaires et les plus simples à comprendre dans ce domaine. D'autres formats existent dans la littérature à l'instar des formats CSC (Compressed Sparse Column), CSB (Compressed Sparse Block) [Buluç et al., 2009] et pleins d'autres formats plus complexes. Mais nous les laissons de côté, car notre étude ne porte pas sur l'optimisation des formats de stockage.

2. https ://github.com/purtroppo/PageRank

Le format COO

Le format de coordonnées (COO) est un schéma standard de stockage des matrices creuses, bien qu'il ne soit pas couramment utilisé pour le SpMV en raison de ses mauvaises performances. La raison principale est que COO ne compresse pas les index de ligne et de colonne, ce qui entraîne une plus grande empreinte mémoire et plus de trafic mémoire pendant un calcul SpMV par rapport à la CSR. Cependant, un algorithme SpMV basé sur COO est intéressant car il comporte des écritures irrégulières. Par conséquent, nous incluons le SpMV basé sur COO en tant que noyau irrégulier que nous utilisons pour étudier la précision de nos modèles prédictifs.

Une matrice creuse A avec N éléments non-nuls est stockée au format COO au moyen de trois tableaux : les indices de ligne $(i_0, i_1, ..., i_{N-1})$, les indices de colonne $(j_0, j_1, ..., j_{N-1})$, et des valeurs non nulles $(a_0, a_1, ..., a_{N-1})$. Les indices non-zéros de ligne n'ont pas besoin des indices colonne, et peuvent donc apparaître dans n'importe quel ordre. Étant donné un vecteur source $x \in \mathbb{R}^n$ et le vecteur de destination $y \in \mathbb{R}^m$, le SpMV $y \leftarrow y + Ax$ est calculé comme suit :

$$y_{i_k} \leftarrow y_{i_k} + a_k x_{i_k}, \text{ pour } k = 0, 1, ..., N-1.$$

Le code suivant montre notre implémentation C naïve de SpMV parallèle à mémoire partagée pour une matrice au format COO.

```
void coo_spmv(int N, int const * i, int const * j,
              float const * a, float const * x, float * y) {
  #pragma omp parallel
  {
    LIKWID_MARKER_START("pageRank");
    #pragma omp for schedule(static)
    for ( int k = 0; k < N; k++) {
      #pragma omp atomic
      y[i[k]] += a[k] * x[j[k]];
    }
    LIKWID_MARKER_STOP("pageRank");
  }
}
```

Notons que des précautions doivent être prises pour éviter les conditions de concurrence dues aux écritures irrégulières. Pour cette raison, une directive atomique *omp* est utilisée dans la boucle. Bien qu'il y ait une surcharge de synchronisation substantielle imposée par les écritures atomiques, et qu'il existe des moyens plus efficaces d'implémenter COO SpMV qui évitent cette surcharge, nous choisissons délibérément la version actuelle pour étudier l'effet des écritures irrégulières.

Le format CSR

Compressed Sparse Row (CSR) est un autre format de stockage standard pour les matrices creuses générales, où les entrées de matrice non nulles et leurs emplacements sont stockés explicitement. Considérons une matrice creuse $A = (a_{i,j}) \in \mathbb{R}^{m,n}$ avec m lignes, n colonnes et N éléments non nuls. En d'autres termes, il existe N paires distinctes d'indices de lignes et de colonnes (i_k, j_k), tels que $a_k := x_{i_k.j_k} \neq 0$. Si les non-zéros sont rangés par ordre croissant selon leurs indices de lignes, alors la matrice **A** peut être représentée au format CSR avec N valeurs non nulles $(a_0, a_1, ..., a_{N-1})$ et indices colonnes $(j_0, j_1, ..., j_{N-1})$, et *m+1* pointeurs de lignes $(r_0, r_1, ..., r_m)$. Les pointeurs de ligne r_i et $r_{i+1} - 1$ sont les indices du premier et dernier non-zéro de la ième ligne respectivement. Autrement dit, un non-zéro a_k appartient à la ligne i_k si $r_{i_k} \leq k < r_{i_k+1}$.

Pour un vecteur source $x \in \mathbb{R}^n$ et le vecteur de destination $y \in \mathbb{R}^m$, le SpMV $y \leftarrow y + Ax$ est calculé comme suit :

$$y_i \leftarrow y_i + \sum_{k=r_i}^{r_{i+1}-1} a_k x_{j_k} \text{ pour } i = 0, 1, ..., m-1.$$

Une stratégie courante pour calculer SpMV en parallèle consiste à partitionner les lignes de la matrice et à répartir les parties entre plusieurs processeurs ou threads. De cette façon, chaque processeur calcule ses valeurs de vecteur destination y_i en fonction du partitionnement en ligne. Les valeurs non nulles a_k, les indices de colonne j_k et les éléments de vecteur de destination y_i sont partitionnés avec les lignes. En revanche, les éléments du vecteur source x_{j_k} peuvent être partagés par plusieurs processeurs, bien que l'étendue de ce partage dépende des indices de colonne j_k.

Le code suivant montre notre implémentation simple en C du SpMV parallèle à mémoire partagée pour une matrice au format CSR.

```
void csr_spmv(int m, int const * r, int const * j,
              float const * a, float const * x, float * y) {
  #pragma omp parallel
  {
    LIKWID_MARKER_START("pageRank");
    #pragma omp for schedule(static)
    for (int i = 0; i < m; i++) {
      float z = 0.0;
      for (int k = r[i]; k < r[i+1]; k++)
        z += a[k] * x[j[k]];
      y[i] += z;
    }
    LIKWID_MARKER_STOP("pageRank");
  }
}
```

Ici, *OpenMP* est utilisé pour paralléliser la boucle externe en répartissant les lignes entre un nombre de *threads* donné. La clause *schedule* de la directive *"omp for"* spécifie que la boucle doit être divisée en morceaux, composés d'itérations de boucle consécutives et que les morceaux sont affectés aux threads de manière circulaire.

La principale différence entre les deux formats c'est que dans le CSR le vecteur d'indice de ligne est compressé. Cela facilite ainsi le stockage des données, réduisant ainsi le trafic mémoire par rapport à COO lors des accès. Cependant, en compressant les lignes ainsi, le format CSR donne lieu à des motifs d'accès mémoire plus complexes qui ne sont pas très pratique pour le parcours de la matrice. Ainsi, pour un code avec peu d'accès à la mémoire, il sera préférable de choisir le format COO.

Une fois nos algorithmes présentés, nous allons voir dans la section suivante une succession d'étapes qui vont nous permettre d'analyser la performance de ces codes que nous avons implémenté.

3.3 Méthodologie

Dans cette section, nous proposons une méthodologie en 5 étapes pour l'analyse des performances d'une application donnée.

3.3.1 Profilage des applications

Les *hot spots* sont particulièrement présents dans les applications de HPC et correspondent généralement aux noyaux de calcul *(kernels)*. Ces zones de codes possèdent un fort potentiel pour l'amélioration de la performance de l'application. Si une application passe 99% de ses cycles dans l'exécution d'une fonction, une amélioration d'un facteur 10 de celle-ci entraînera une amélioration du même facteur de l'application. Le travail du programmeur est donc d'identifier et d'accélérer ces parties en priorité.

Les applications réelles utilisées en production dépassent souvent les dizaines de milliers de lignes de codes. Porter et optimiser la totalité d'une application serai complexe et contre-productif. De plus, la performance de chaque kernel peut être limitée par des parties différentes du matériel (*memory bound, compute bound, IO bound, etc.*), il est donc nécessaire de les porter individuellement sur différentes plateformes.

L'objectif de cette étape est d'identifier ces zones clés du code et de modéliser leur performance en fonction du trafic des données dans la hiérarchie mémoire. Des outils existent dans la littérature pour réaliser ce travail de profilage à l'instar de : *perf, Oprofile, gprof, etc.* Il est nécessaire de découper son programme en plusieurs fonctions afin que ces outils marchent correctement. On peut aussi procéder de manière *ad-hoc* en instrumentant directement le code et en mesurant les temps d'exécution des parties suspectées du code.

3.3.2 Modélisation analytique des Kernels

Un modèle de performance analytique est une description simplifiée des interactions entre le logiciel et le matériel ensemble avec une recette pour générer des prédictions de temps d'exécution. Un tel modèle doit être simple pour être maniable, mais aussi suffisamment élaboré pour produire des prédictions utiles.

Les modèles purement analytiques (c'est-à-dire les premiers principes ou la boîte blanche) sont basés sur des détails techniques connus du matériel et quelques hypothèses sur la façon dont le logiciel s'exécute. L'exemple classique d'un modèle de boîte blanche est le modèle du **Roofline** de [Williams et al., 2009] pour la prédiction des performances des boucles (noyaux d'exécution). La précision de telles prédictions dépend essentiellement de la fiabilité des détails de bas niveau. Un manque de prévision remet en question les hypothèses sous-jacentes et une fois corrigé, conduit souvent à une meilleure compréhension.

La majorité des codes étant limitée par le débit mémoire, nous utilisons des modèles basés sur le volume de données qui traverse le contrôleur mémoire. Étant sur une architecture multicœur et notre application **PageRank** qui peut être parallélisé, nous analysons également le temps d'exécution du programme ainsi que la scalabilité de l'application sur plusieurs *threads*. Nous analysons donc l'activité du bus mémoire comme élément de la microarchitecture.

Pendant cette étape, nous avons utilisé les trois modélisations suivantes :

— Le modèle du trafic mémoire qui est indépendant de l'architecture ;

— Le modèle **SMM** pour prédire le temps optimal ;

— La loi d'Amdahl pour prédire l'accélération maximale.

3.3.3 Instrumentation du code source

L'instrumentation du code source dans l'analyse de performances consiste à ajouter les instructions supplémentaires dans le code pour délimiter la région spécifique du code à analyser. Ainsi, les instructions ajoutées dépendent du but à atteindre par l'instrumentation, à savoir :

— Le profilage de code utilise l'instrumentation pour enregistrer les fonctions appelées et le temps passé dans chacune d'elles, afin d'identifier les parties de code à optimiser.

— Le tissage dans la programmation orientée aspect modifie le binaire généré lors de la compilation en ajoutant des capacités supplémentaires au programme.

— La détermination de la couverture de code en utilisant l'instrumentation de code pour enregistrer les instructions exécutées et les chemins empruntés lors de l'exécution.

— La détection des problèmes dans l'utilisation de la mémoire *(fuite de mémoire, déréférencement de pointeur NULL, etc.)* dans un programme en l'instrumentant et en ajoutant des vérifications à l'exécution.

Pour notre méthodologie, nous utilisons l'instrumentation du code avec l'*API MakerAPI* de *LIKWID* pour délimiter les kernels d'exécution de notre code. Cela permet par la suite d'effectuer la mesure de performance de la région du code ainsi délimitée. Le code de **l'annexe A.1** nous montre une instrumentation pour notre application **GraFC2T2** pour la mesure des performances de la région calculant la matrice de similarité de Jaccard.

3.3.4 Mesure des performances réelles

Grâce aux instrumentations de code de l'étape précédente, nous allons mesurer les performances réelles de l'application (i.e de la région de code analysé) à l'aide des compteurs matériels présents dans l'architecture.

Ici, nous exécutons notre application avec le module *likwid-perfctr* de *LIKWID* avec le mode *Perf Events* pour accéder aux compteurs matériels via le registre *msr*. Nous utilisons le groupe d'événement **"MEM"** pour mesurer le trafic mémoire en termes de volume de données et bande passante mémoire.

Pour la performance en temps d'exécution, nous utilisons la fonction **clock_gettime** du système *Linux* pour les mesures des temps séquentiel et parallèle du code. Cette fonction a une précision allant jusqu'à la nanoseconde.

3.3.5 Comparaison et interprétation des résultats

Une fois qu'on a fait la mesure des performances de notre application lors de son exécution, nous allons faire une comparaison avec les prédictions des modèles précédent pour voir la précision de celui-ci. Le but de cette étape est de voir les écarts entre les deux valeurs et de pouvoir justifier ces observations grâce à notre connaissance de l'architecture et de l'application. Cette étape nous permet également de confirmer des intuitions ou des hypothèses de départ sur le comportement de nos applications grâce à l'interprétation des chiffres. À l'issue de cette étape, de nouvelles hypothèses peuvent être établies.

Ici, nous avons comparé les architectures *Intel Skylake 6130* et *ARM TX2*. Nous avons également comparé les prédictions de nos modèles sur les 2 architectures pour notre application **PageRank**.

Une fois nos algorithmes connus, implémentés et notre méthodologie élaborée, nous pouvons passer aux expérimentations afin d'étudier le comportement de nos applications.

Chapitre 4: Étude expérimentale

Dans ce chapitre, nous présentons les expérimentations qui permettent d'évaluer les modèles de performance du chapitre précédent. Pour cela, nous présentons d'abord notre protocole expérimental. Ensuite, nous présentons les résultats de profilage qui nous ont guidés dans l'analyse et qui nous ont poussés à nous concentrer sur l'application PageRank. Nous continuons en présentant les résultats de la performance du compilateur. Après, nous avons les résultats monocœur et nous terminons par la performance multicœurs.

4.1 Protocole expérimental

Dans cette section, nous présentons un bref aperçu du cadre des expérimentations qui ont été réalisées pendant nos travaux. Ces exécutions ont été menées sur la plateforme de calcul **Grid'5000**. Cette dernière est un *testbed* flexible et à grande échelle pour la recherche expérimentale dans tous les domaines de l'informatique, avec un accent sur le calcul parallèle et distribué, y compris le Cloud, le HPC, le Big Data et l'IA. Cette plateforme, basée en France, nous a été accessible à travers la connexion à distance SSH depuis nos laptops au Cameroun. Nous décrivons donc les deux architectures sur lesquelles nous avons mené notre étude ainsi que les jeux de données.

4.1.1 Machines d'exécution

Ici, nous présentons juste les caractéristiques des processeurs qui nous aident dans l'interprétation des résultats ou qui peuvent avoir une influence sur les performances du programme. Dans le tableau 4.1, nous pouvons noter que de façon générale *Marvell Thunder X2* a de meilleures caractéristiques comparé à *Intel Xeon Skylake 6130*. On peut donc imaginer de meilleures performances sur TX2.

Noms du cluster	dahu	pyxis
Processeur	Xeon Skylake 6130	Thunder X2
cœurs(threads)	16(32)	**32(128)**
cache L1	64Kb, 8-way	64Kb, 8-way
cache L2	**1024Kb, 16-way**	256Kb, 8-way
cache L3	22Mb, 11-way	**32Mb, fully**
exclusivité	**tous**	L3 exclu L2
DRAM	DDR4-2666MTS, 6 cannaux	**DDR4-2666MTS, 8 cannaux**
Bandwidth DRAM	128Gb/s	**170Gb/s**
OS	Linux 5.11	**Linux 5.15**

TABLE 4.1 – Informations sur les plateformes de test sur *Grid5000*. Les éléments en gras indiquent la meilleure plateforme selon la caractéristique associée. Pour plus de détails sur ces éléments, voir la section **1** du **chapitre 1** qui décrit les architectures étudiées.

NB : Pour nos serveurs d'expérimentations, les processeurs sont montés en double CMP suivant l'architecture *NUMA (Non Uniform memory Access)*. Donc les tailles de cache L3, *Bandwidth DRAM* (Bande passante théorique maximale) et nombre de cœurs sont doublés.

4.1.2 Jeux de données

Ici nous présentons les jeux de données qui ont été utilisés dans les expérimentations avec nos deux applications. Nous avons des jeux de données de taille variable afin de mieux évaluer le comportement des performances de l'application. Nous utilisons des jeux de données publiques afin de nous assurer qu'ils sont bien construits et éviter les résultats biaisés. Le tableau 4.2 nous présente une synthèse des informations sur les jeux de données de l'application PageRank et 4.3 présente ceux utilisés pour l'application GraFC2T2.

noms	nombre de noeuds	nombre d'arêtes	taille sur disque	sparsité
soc-LiveJournal1	4,847,571	68,993,773	1,080MB	99.999%
web-BerkStan	685,230	7,600,595	110MB	99.997%
web-Stanford	281,903	2,312,497	32MB	99.998%

TABLE 4.2 – Les jeux de données pour l'application **pageRank** en C sont des graphes dirigés simples. Disponibles sur : snap.stanford.edu

noms	utilisateurs	produits	contenus	votes	sparsité
ciao-data	890	9 084	6	12 753	99.970%
epinions	1 999	24 861	24	28 399	99.997%

TABLE 4.3 – Les jeux de données pour l'application **GraFC2T2** en C++. Plus de détails sur les fichiers de confiances sur : github.com/nzekonarmel/GraFC2T2

4.2 Extraction des noyaux d'exécution dans GraFC2T2

Nous avons commencé par le profilage de notre application de base **GraFC2T2**. Cela nous a permis comme nous pouvons le constater avec le tableau 4.4 de comprendre que le calcul du **PageRank** et celui de la similarité de *Jaccard* sont les principaux noyaux de calcul (*hots spot*) de notre application. La similarité de *Jaccard* domine sur le pourcentage du temps d'exécution. On peut constater que plus le jeu de données est grand, plus la proportion du temps d'exécution de **jaccard** augmente (plus de 80% avec le jeu de données **epinions** sur les 2 architectures). Cela se justifie, car son nombre d'appels est plus élevé comparé au nombre d'appel de la partie **PageRank**. De plus, le calcul du PageRank dépend d'un critère d'arrêt qui dans notre cas avec les jeux de données utilisés s'arrête juste après 20 itérations. Pour un jeu de données avec une centaine d'itérations, PageRank pourra dominer sur Jaccard.

Le calcul de *Jaccard* étant spécifique à l'enrichissement pour améliorer la qualité de recommandation, nous choisissons de continuer avec **PageRank** dans les analyses car c'est un noyau plus générique des applications de recommandation à base de graphes. Les tableaux 4.4 et 4.5 nous montrent les résultats quantitatifs qui ont guidé notre choix. Nous avons compilé l'application avec **-O3** et **g++**

/	Skylake (total)	TX2 (total)	Skylake (Jaccard)	TX2 (Jaccard)
ciao-data	22s	60s	12s	36s
epinions	263s	740s	215s	644s

TABLE 4.4 – Profilage de l'application *GraFC2T2* avec le graphe biparti simple (BIP). Temps total de tous les appels

/	**PageRank**	**Jaccard**
ciao-data	550	395 605
epinions	755	1 997 001

TABLE 4.5 – Profilage de l'application *GraFC2T2* avec le graphe biparti simple (BIP). Le nombre d'appels de Jaccard est très élevé et justifie le temps d'exécution de cette partie. Nous l'avons instrumenté avec le calcul de la matrice de note binaire (voir **Annexe A.1**), car Jaccard en dépend et elle occupe un temps non-négligeable que nous n'avons pas détaillé ici.

4.3 L'impact du niveau d'optimisation du compilateur

Il est possible d'optimiser un code lors du processus de compilation. En plus, nos modèles supposent que le compilateur est parfait. Ce qui nous amène à choisir la meilleure optimisation possible pour la compilation. Cependant, vu que le nombre d'options d'optimisation est important, nous ne considérons ici que les options d'optimisation de base **-Ox**. Il existe quatre niveaux d'optimisations :

— **O0** : Ce niveau n'effectue aucune optimisation.

— **O1** : Ce niveau optimise la taille du code et la localité des données. Il peut augmenter les performances d'application ayant un grand nombre de ligne de code et plusieurs branchements, mais où le temps d'exécution n'est pas dominé par le temps des boucles.

— **O2** : C'est le niveau par défaut que l'on retrouve sur les compilateurs modernes. Il optimise la vitesse d'exécution du programme.

— **O3** : Il reprend les optimisations de **-O2** tout en étant plus agressif sur l'optimisation des boucles, des accès mémoire et le *préchargement*. Cette optimisation ne donne pas forcément de meilleures performances que l'option **-O2** et est principalement recommandée pour les applications ayant un grand nombre de boucles, une utilisation massive des nombres à virgule flottante ou utilisant des données de grande taille.

	TX2				**Skylake**			
	COO		**CSR**		**COO**		**CSR**	
Niveaux	**gcc**	**clang**	**gcc**	**clang**	**gcc**	**clang**	**gcc**	**clang**
O0	**89.2**	**90.1**	**79.1**	**79.5**	**46.1**	**45.5**	**38.1**	**38.5**
O1	58.9	55.5	48.0	48.1	25.7	25.5	21.8	22.1
O2	57.8	55.3	46.0	46.2	25.6	25.3	21.6	21.8
O3	57.9	55.4	45.8	46.3	25.3	25.5	21.5	21.6

TABLE 4.6 – Récapitulatif des résultats sur les optimisations du compilateur avec le Jeu de données LiveJournal1 sur l'application **PageRank**. Les résultats sont en seconde.

Nous avons choisi le jeu de données *LiveJournal1* ici pour permettre l'exécution sur plusieurs secondes. Le tableau 4.6 montre le résultat de l'expérimentation et nous pouvons remarquer que :

— Globalement, le comportement est identique entre les compilateurs *gcc/g++* et *clang/clang++* sur nos deux architectures Intel et ARM.

— Nous pouvons le confirmer, les optimisations pour PageRank sont moins variables comparées à celles de *GraFC2TC*. Ce qui se justifie car *GraFC2TC* a plus de boucles, de branchements conditionnels et de lignes de code.

— L'optimisation est importante pour accélérer l'exécution de l'application et se rapprocher du compilateur parfait. Notre code a besoin d'optimisation comme nous pouvons le voir avec l'exécution -O0 qui donne la plus mauvaise performance et très éloignée de celle des autres niveaux.

— Les niveaux -O2 et -O3 ont des résultats presque identiques pour *PageRank*. Ce qui se justifie par le nombre de lignes de code qui est faible et moins de boucles.

Grâce à ces chiffres, nous pouvons choisir *gcc/g++* pour la suite car il donne des meilleures performances que *clang/clang++* sur les grosses applications.

4.4 Modélisation, évaluation et analyse monocœur

Dans cette section, nous utilisons les versions séquentielles de notre application PageRank pour comparer les prédictions de nos modèles aux résultats d'expérimentations. En effet, l'application **PageRank** a été utilisé dans le but de restreindre l'analyse à l'algorithme **PageRank** qui est le principal noyau de calcul des systèmes de recommandation que nous avons étudiés.

4.4.1 Profilage de l'application PageRank

Avant de continuer des analyses plus poussées sur notre application **PageRank**, nous faisons d'abord un profilage pour extraire les noyaux d'exécutions. Le tableau 4.7 nous montre les temps d'exécution des deux grands noyaux d'exécution. Nous pouvons voir en **annexe B.1** le code de la lecture des données dans le fichier qui domine sur TX2. On peut aussi confirmer la mauvaise performance du format COO sur les deux architectures pour nos implémentations par son temps d'exécution élevé.

Processeur		**Skylake 6130**	**TX2**
temps lecture données		10s	30s
temps SpMV	COO	15s	28s
	CSR	12s	15s
autres		1s	1s
total		**23s(26s)**	**46s(59s)**

TABLE 4.7 – Profilage de l'application *PageRank*.

Ici, *"temps lecture données"* représente le temps pour lire les données du graphe depuis le fichier sur le disque dur ; *"SpMV"* c'est pour le calcul de la multiplication matrice creuse vecteur et *"autres"* c'est pour le reste du code de l'application. Le temps total c'est le temps mis par l'application pour une exécution complète en utilisant le format CSR. Le temps total entre parenthèses c'est celui mis en utilisant le format COO.

4.4.2 Limitation de performance pour nos deux noyaux

Une fois les noyaux extraits, il est nécessaire de qualifier les limitations de performances sur l'architecture. Rappelons qu'il est important d'analyser chaque noyau individuellement, car au sein d'une même application, la performance de chaque kernel peut être limitée par des parties différentes du matériel. Ainsi, nous analysons le débit mémoire pour savoir si notre application est **MEMORY BOUND** (performance limitée par la mémoire) ou **COMPUTE BOUND** (limitée par le système de calcul). Cela peut se faire de façon théorique en utilisant le modèle de Roofline

présenté précédemment ou bien de façon plus pratique en exécutant réellement l'application avec un outil d'analyse de performance pour mesurer la performance maximale réelle. Nous choisissons de faire des mesures pratiques, car le modèle du Roofline ne tient pas en compte les détails techniques d'implémentation qui peuvent impacter considérablement les performances.

Ainsi, grâce au tableau 4.8, nous pouvons effectivement confirmer que nos deux noyaux sont très limités par la performance mémoire car nous n'atteignons pas les 1/8 ème de la valeur monocœur mesuré avec STREAM.

Processeur	Skylake 6130	Thunder X2
MEM Data Volume	37GB	35GB
MEM Bandwidth	3.12GB/s	2.3GB/s

TABLE 4.8 – Bandes passantes mémoires pour le calcul SpMV dans l'application PageRank, mesurées avec LIKWID. Sur nos deux architectures la Bande passante reste très faible pour des volumes de trafic très élevé.

Processeur	Skylake 6130	Thunder X2
MEM Data Volume	1.92GB	1.96GB
MEM Bandwidth	0.198GB/s	0.066GB/s

TABLE 4.9 – Bandes passantes mémoires pour le chargement des données dans l'application PageRank, mesurées avec LIKWID. On note que cette Bande passante est plus faible.

Nous pouvons donc conclure que nos noyaux de calcul sont très MEMORY BOUND.

Afin de mieux comprendre les performances, nous allons continuer dans les sections suivantes avec des analyses de trafic de données du $SpMV$ et de la scalabilité pour l'application. Nous laissons de côté l'analyse de la partie lecture des données, car cela relève des accès disque alors que nous avons délimité notre travail à l'analyse de la partie mémoire (i.e des accès mémoire).

4.4.3 Analyse des trafics mémoire

Nous analysons ici les volumes de données qui traversent le contrôleur mémoire et nous faisons des comparaisons avec les prédictions de notre modèle des empreintes mémoire. Nous utilisons également la bande passante mesurée afin de corréler les volumes de données transférées et leurs taux de traitement. Pour cela, nous utilisons le modèle des empreintes mémoire.

Modélisation du volume de trafic mémoire

Cette première modélisation se fait à partir du code source en comptant les accès et en les multipliant par la taille en octet du type de données accédé. Il est important de multiplier le résultat par le nombre total d'itérations qui n'est malheureusement pas connu d'avance, mais que l'on peut fixer une valeur maximale. Nous avons pour les codes de la section **3.2.4** du chapitre précédent les modèles du tableau 4.10 :

noyau	SpMV_COO	SpMV_CSR
Volume	$3(4N) + 2(4m) = 12N + 8m$	$8N + 12m$

TABLE 4.10 – Modèle du volume de données pour les deux versions du noyau de calculs. Ce tableau contient le nombre d'octets qui vont traverser la mémoire durant le calcul. Plus précisément, c'est une borne inférieure de la quantité de données à transiter sur le bus mémoire. Noter que nous considérons les entiers et flottants sur 32 bits.

Avec : **"N"** le nombre d'éléments non nuls dans la matrice creuse ; **"m"** c'est le nombre de colonnes. On peut déjà imaginer que le trafic sera plus élevé avec le format COO puisque **N** est généralement plus grand que **m**.

NB : ce premier modèle est indépendant de l'architecture et a juste besoin des informations sur l'application et les données utilisées.

Sur la figure 4.1, nous pouvons observer que TX2 se rapproche plus des prédictions en termes de volume de données comparé à Skylake pour les deux formats. Les prédictions sont plus faibles que les volumes mesurés, mais se rapprochent. Cela montre que la réutilisation des données dans le cache est assez bien gérée sur les deux architectures. Cependant, on note que sur Skylake les volumes sont les plus élevés. Cela montre que la gestion des caches sous TX2 est meilleure, car on a moins d'accès à la mémoire centrale selon les volumes de données qui circulent sur le contrôleur mémoire. De plus, nous pouvons mieux confirmer la mauvaise performance du format COO telle qu'énoncé en section **3.1.4** avec le volume de données élevé pour ce format.

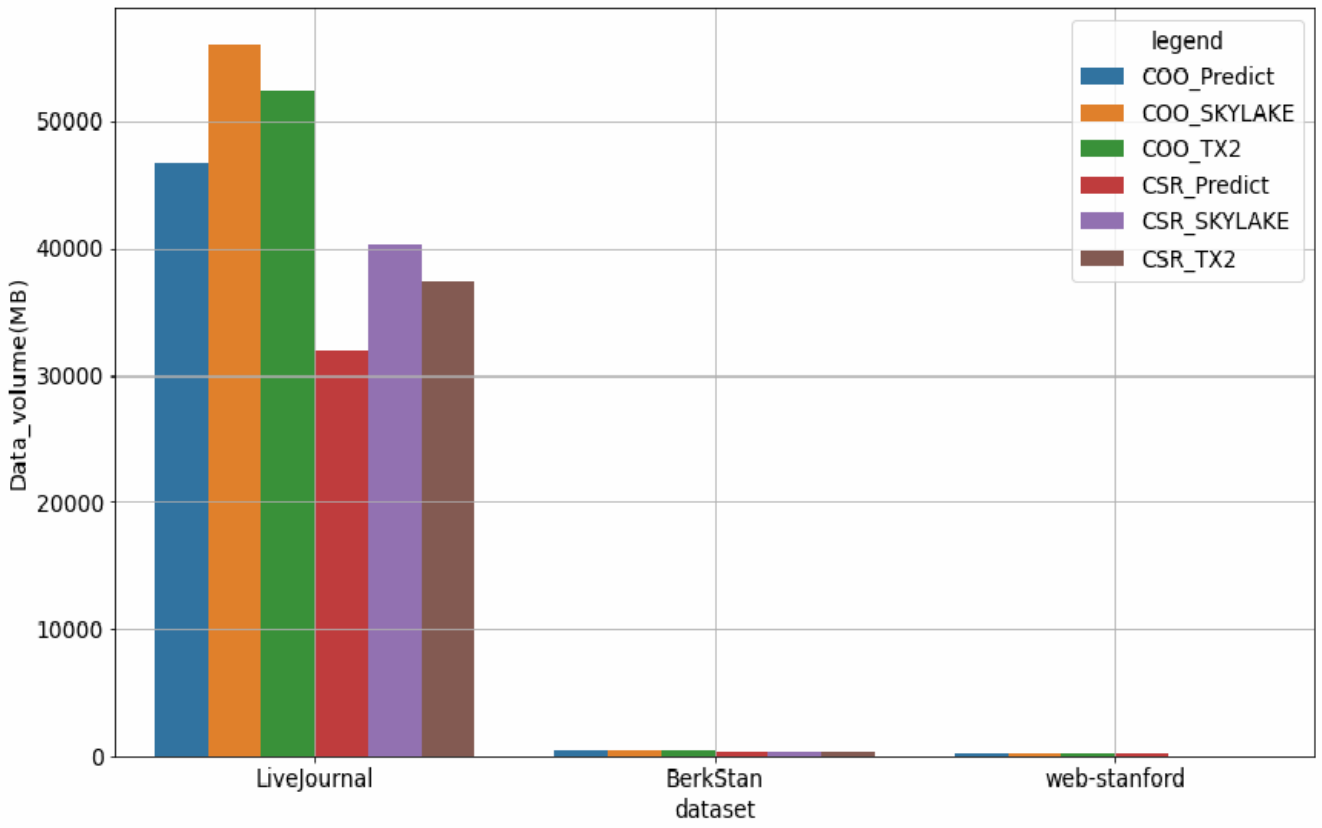

FIGURE 4.1 – Analyse du volume de trafic des données pour les différents Jeux de données sur nos deux architectures de test.

Nous observons un comportement assez similaire avec les jeux de données de petites tailles (**Web-Bekstan** et **Web-Stanford**) sur la figure 4.2. Sauf que cette fois-ci, les prédictions du modèle sont plutôt plus élevées. Cela démontre qu'avec des jeux de données de petite taille, il y a une meilleure réutilisation des données en cache (les données tiennent plus facilement en caches) et donc moins d'accès à la mémoire centrale. Nous pouvons ainsi dire qu'un volume de trafic

élevé est cause de mauvaise performance sur l'architecture. Cependant, nous constatons avec le jeu de données **Web-Bekstan** un comportement différent pour le format COO. Après plusieurs exécutions, les résultats étaient toujours pareils. Cela montre que le jeu de données a un grand impact pour valider les prédictions sur les architectures.

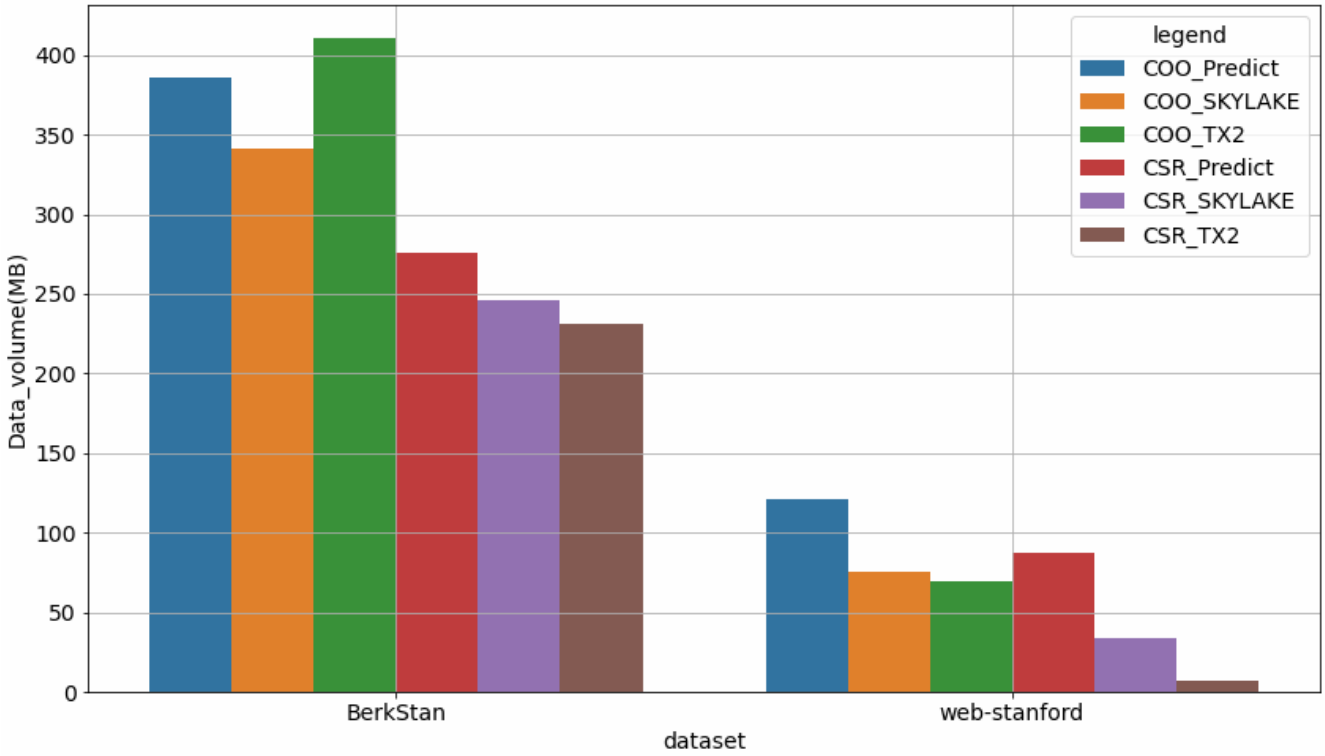

FIGURE 4.2 – Analyse du volume de trafic des données pour **Web-Bekstan** et **Web-Stanford**. Zoom sur la figure 4.1.

Bande passante mémoire

Nous utilisons ici la bande passante mémoire pour analyser la corrélation entre le volume de données transféré et le débit de traitement ; car on peut dans certains cas obtenir une bande passante faible parce qu'il n'y a pas assez de données pour circuler simultanément sur le bus mémoire. Cela supposerait que dans notre cas le format *COO* avec le jeu de donnée **LiveJournal1** aura la meilleure bande passante. Malheureusement, sur la figure 4.3 ce n'est pas le cas. C'est plutôt le format *CSR* qui est meilleur. Nous comprenons que pour nos applications, la bande passante n'est pas fonction du volume de données. Le format *COO* est plus irrégulier et non-prévisible que *CSR* à cause de la non-compression des lignes qui augmente les lignes de caches lors du transfert des données. Voilà une raison de plus qui justifie la mauvaise performance du *COO*.

4.4.4 Analyse du temps d'exécution

Modèle SMM

Ici, nous utilisons le modèle SMM avec la bande passante monocœur maximale de l'architecture en tenant compte du nombre d'itérations. Nous utilisons la bande passante mono-cœur pour nous rapprocher le plus des suppositions du modèle sur le nombre de cœurs car nous modélisons ici le temps séquentiel de l'application. Les valeurs ainsi mesurées avec *STREAM* sont respectivement

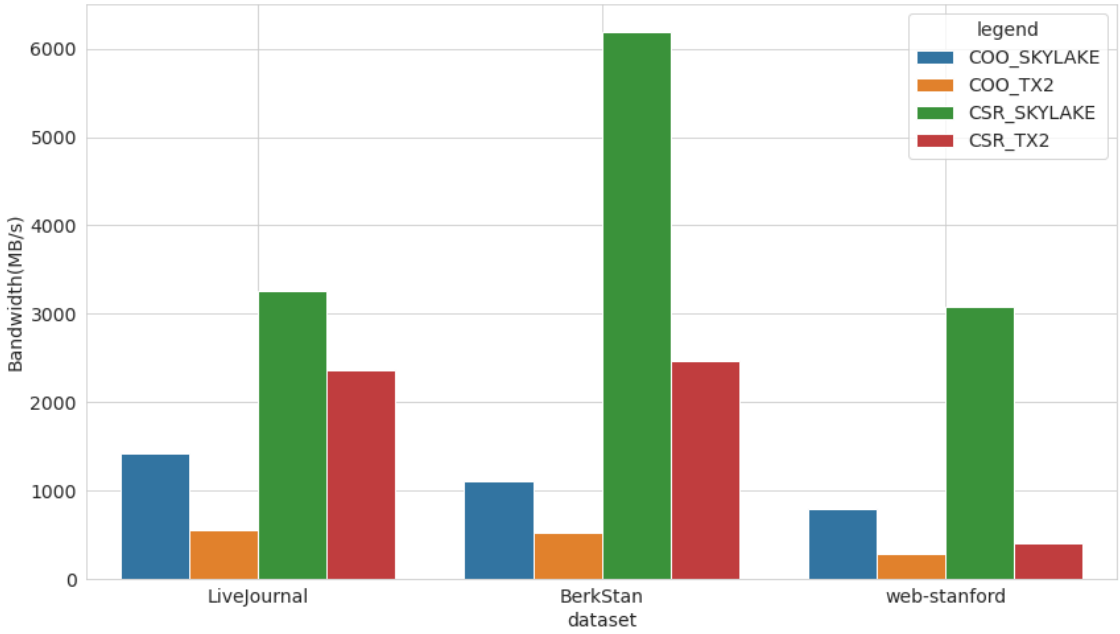

FIGURE 4.3 – Bande passante mémoire pour le calcul du SpMV dans l'application PageRank. Nous notons des bandes passantes très faibles (environ 100 fois plus petites que la bande passante théorique maximale sur TX2 et Skylake.

de **17.1GB/s** sur TX2 et **16.3GB/s** sur Skylake. Le tableau 4.11 nous présente une comparaison des prédictions de notre modèle avec les temps réels d'exécution. Nous avons les temps mesurés entre les parenthèses pour chaque jeu de données et chaque format sur nos architectures de test.

Nous rappelons ici la formule utilisée pour calculer chacune des valeurs théoriques à partir du modèle :

$$TEMPSoptimal = \frac{DATAsize}{MEMORYpeak}$$

Ainsi, pour la valeur de SpMV_COO sur TX2 par exemple et avec LiveJournal1, on a un volume de trafic total de 46.802GB pour les 54 itérations d'exécution. En divisant cette valeur par notre bande passante de 17.1GB/s, on obtient environ 2.732 secondes comme dans le tableau 4.11. A noter qu'ici **DATAsize** prend déjà en compte le nombre d'itérations de PageRank.

Nous remarquons que les prédictions sont très éloignées des résultats réels. Cela est sûrement dû à cause des nombreuses suppositions que le modèle SMM admet et que nous avons énoncé en section **2.3** du **chapitre 2**. Ainsi, on peut comprendre que plusieurs autres facteurs sont à prendre en compte pour mettre sur pied un modèle assez proche de la réalité.

Dataset	**LiveJournal**	**Berk-Stanford**	**Web-Stanford**
Processeur	**TX2**		
SpMV_COO	2.732(27.102)	0.090(7.283)	0.028(2.102)
SpMV_CSR	1.872(16.242)	0.064(3.240)	0.020(0.980)
Processeur	**Skylake**		
SpMV_COO	2.871(25.120)	0.029(3.800)	0.067(1.441)
SpMV_CSR	1.964(22.000)	0.095(2.230)	0.021(0.853)

TABLE 4.11 – Application du modèle SMM sur nos 2 architectures pour estimer le temps théorique optimal. TX2 promet le meilleur temps d'exécution de façon théorique, ceci grâce à sa bande passante.

4.5 Modélisation, évaluation et analyse multicœurs

Ici, nous analysons la scalabilité (passage à l'échelle) sur le nombre de *threads*. Nous avons réalisé une version parallèle de notre application en utilisant **OpenMP**. Nous utilisons la loi d'Amdahl pour prédire le temps d'exécution en fonction du nombre de *threads*.

Modélisation

Reprenons la loi d'Amdahl du chapitre 2, nous avons :

$$S = (\frac{1}{r_s + \frac{r_p}{n}})$$

où r_s est le pourcentage de la partie séquentielle du programme, r_p correspond à celui de la partie parallèle du programme et n est le nombre de processeurs utilisé dans la version parallèle.

Ainsi, grâce à notre profilage de l'application PageRank précédent, on a par exemple pour le format COO sur TX2, un temps total de 57 secondes pour un temps de calcul du *SpVM* (notre région parallèle) de 28 secondes. Donc $r_p = 47\%$ pour ce cas. Suivant cette démarche, nous avons le tableau 4.12 qui présente les valeurs utilisées pour calculer notre accélération S sur le jeu de données *LiveJournal1*. Nous avons le temps total séquentiel en secondes pour chaque format de données sur chaque architecture et nous avons entre parenthèses le pourcentage de la région parallélisable. Nous pouvons constater que le code est plus parallélisable sur Skylake comparé à TX2.

processeur	**COO**	**CSR**
Skylake	25(60%)	22(55%)
TX2	57(47%)	46(34%)

TABLE 4.12 – Temps séquentiel et pourcentage de temps parallélisable (entre parenthèses) pour la loi d'*Amdahl*

Scalabilité sur le nombre de threads

Sur la figure 4.4, nous pouvons remarquer que sur **Skylake** les prédictions se rapprochent plus du temps réel comparé à **TX2** dont les prédictions du format *COO* sont assez éloignés des valeurs réelles. Sur *TX2*, lorsque le nombre de *threads* dépasse 8, le temps prédit du *COO* devient plus élevé que le temps mesuré. Par contre, sur **Skylake** ce temps prédit est en général plus petit que le temps mesuré. Cela est dû à la très mauvaise performance du format COO pour la version séquentielle.

En plus, nous pouvons confirmer que la loi d'**Amdahl** ne tenant pas en compte les frais de parallélisation n'est pas très pratique avec un nombre de *threads* petit dans notre cas pour le format *COO*. Ce qui démontre que COO a besoin d'une bonne parallélisation pour rattraper la performance de CSR sur les deux architectures.

Les courbes de la légende se terminant par *"Amdahl"* représentent les temps d'exécution obtenus en divisant le temps séquentiel du programme par la valeur de l'accélération pour le nombre de *threads* utilisé. Nous choisissons d'afficher les temps d'exécution sur le graphe pour permettre une meilleure visibilité des données par rapport aux valeurs plus petites en termes d'accélération.

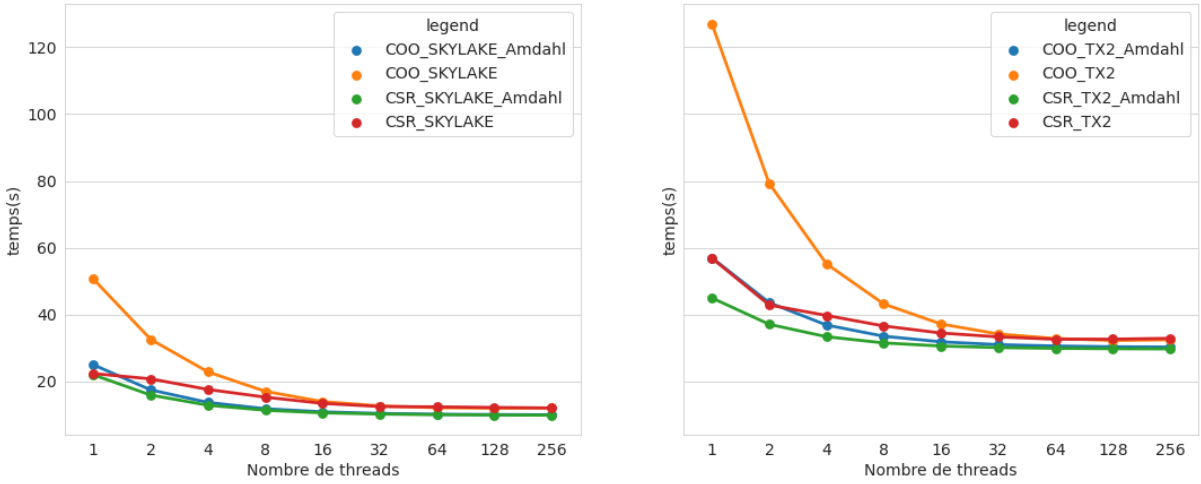

FIGURE 4.4 – Évolution du temps d'exécution de l'application PageRank en fonction du nombre de threads.

Conclusion et perspectives

Dans ce travail de mémoire, l'objectif était de modéliser les applications de recommandation sur les processeurs **ARM ThunderX2** afin de prédire leurs performances. Les architectures *ARM* ont démontré grâce aux travaux de la littérature leurs compétitivités en termes de performance à ceux d'*Intel* et cela pour une consommation énergétique réduite [Hammond et al., 2019]. Mais, ces travaux portent plus sur la modélisation des applications HPC de calcul scientifique qui présentent généralement des schémas d'accès mémoire prévisible. Malheureusement, nos applications présentent des schémas d'accès à la mémoire centrale irréguliers et non-prévisible. Ce qui rend plus difficile la prédiction des performances. Nous nous sommes donc concentré sur la prédiction du trafic mémoire ainsi que du temps d'exécution sur ce processeur en comparaison au processeur **Xeon Gold Skylake 6130** d'*Intel*.

Nous avons vu que les performances de nos deux applications sont limitées par la bande passante mémoire sur ces deux architectures. Nos modèles prédictifs donnent des résultats assez précis sur les volumes de données qui vont transiter sur la mémoire centrale **DRAM**. Pour la modélisation du temps d'exécution, la loi d'**Amdahl** donne des bonnes prédictions lorsque le nombre de *threads* devient élevé (i.e plus de 8 *threads*). Cela montre que la parallélisation devient plus efficace lorsque le nombre de threads de *CPU* devient plus grand. Par contre, le modèle SMM ne marche pas bien pour nos applications. Cela se justifie par les suppositions de ce modèle qui ne prend pas en compte certains aspects de performance à savoir le compilateur et la gestion du cache.

Nous avons commencé par comparer les performances de deux compilateurs différents sur les options d'optimisation de base (*gcc* et *clang*), afin de choisir le meilleur compilateur pour nos applications. Nous avons constaté que le compilateur *gcc* est en général meilleur que *clang* et le niveau O3 permet d'avoir les meilleures performances. Ensuite, grâce au profilage de code, nous avons identifié *PageRank* comme notre principal noyau de calcul et que pour certains jeux de données comme **LiveJournal1**, les accès disque peuvent dominer sur le calcul. Mais *Skylake* offre une meilleure gestion des accès disque sur **PageRank** comparé à *TX2*. Ensuite, nous avons constaté que *TX2* en général offre une meilleure gestion du cache comparé à *Skylake* grâce au volume de données réduit que nous avons mesuré. Cependant, le temps d'exécution sur *Skylake* reste réduit. Enfin, nous avons constaté que *TX2* arrive à rattraper les performances de *Skylake* sur la région parallèle de notre **PageRank** lorsque le nombre de *threads* augmente.

En perspective, pour comprendre davantage les différences de performance entre les deux compilateurs et sur ces deux architectures, nous proposons de creuser les détails du code assembleur généré pour identifier les différentes optimisations appliquées. Nous projetons également de modéliser les performances des autres processeurs ARM à l'instar du processeur **A64FX** de *Fujutsu* qui présente un système de caches plus simplifié avec juste deux niveaux de caches. On pourra donc y appliquer le modèle **ECM** qui n'a pas pu être appliqué ici à cause de la complexité des caches de TX2.

Bibliographie

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA. Association for Computing Machinery.

[Bucek et al., 2018] Bucek, J., Lange, K., and Kistowski, J. (2018). Spec cpu2017 : Next-generation compute benchmark. *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*.

[Buluç et al., 2009] Buluç, A., Fineman, J. T., Frigo, M., Gilbert, J. R., and Leiserson, C. E. (2009). Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, page 233–244, New York, NY, USA. Association for Computing Machinery.

[Calore et al., 2020] Calore, E., Gabbana, A., Schifano, S., and Tripiccione, R. (2020). Thunderx2 performance and energy-efficiency for hpc workloads. *Comput.*, 8 :20.

[Calore et al., 2018] Calore, E., Mantovani, F., and Ruiz, D. (2018). Advanced performance analysis of hpc workloads on cavium thunderx. *2018 International Conference on High Performance Computing and Simulation (HPCS)*, pages 375–382.

[Charif-Rubial et al., 2014] Charif-Rubial, A. S., Oseret, E., Noudohouenou, J., Jalby, W., and Lartigue, G. (2014). Cqa : A code quality analyzer tool at binary level.

[Djoudi et al., 2005] Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.-T., and Jalby, W. (2005). Maqao : Modular assembler quality analyzer and optimizer for itanium 2.

[Dongarra et al., 2003] Dongarra, J., Luszczek, P., and Petitet, A. (2003). The linpack benchmark : past, present and future. *Concurrency and Computation : Practice and Experience*, 15.

[Hager et al., 2016] Hager, G., Treibig, J., Habich, J., and Wellein, G. (2016). Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency and Computation : Practice and Experience*, 28 :189 – 210.

[Hammond et al., 2019] Hammond, S., Hughes, C., Levenhagen, M., Vaughan, C., Younge, A., Schwaller, B., Aguilar, M. J., Pedretti, K., and Laros, J. (2019). Evaluating the marvell thunderx2 server processor for hpc workloads. *2019 International Conference on High Performance Computing and Simulation (HPCS)*, pages 416–423.

[Israel and Gideon,] Israel, H. and Gideon, S. Intel architecture code analyzer. https://software.intel.com/content/www/us/en/develop/articles/intel-architecture-code-analyzer.html. Acccessed : 28 April 2022.

[Jaccard,] Jaccard, P. The distribution of the flora in the alpine zone.1. *New Phytologist*, 11 :37–50.

[Levon and Elie, 2004] Levon, J. and Elie, P. (2004). Oprofile : A system profiler for linux.

[LLVM,] LLVM. Llvm machine code analyzer. https://llvm.org/docs/CommandGuide/llvm-mca.html. Acccessed : 28 April 2022.

[McCalpin, 1995] McCalpin, J. D. (1995). Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25.

[Nzekon Nzeko'o, 2019] Nzekon Nzeko'o, A. J. (2019). *Système de recommandation avec dynamique temporelle basée sur les flots de liens*. Theses, Sorbonne Université ; Université de Yaoundé I.

[Page et al., 1999] Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking : Bringing order to the web. In *WWW 1999*.

[Pourroy, 2020] Pourroy, J. (2020). *Calcul Haute Performance : Caractérisation d'architectures et optimisation d'applications pour les futures générations de supercalculateurs*. PhD thesis, Université Paris-Saclay.

[Report et al., 2013] Report, S., Dongarra, J., and Heroux, M. (2013). Toward a new metric for ranking high performance computing systems.

[Treibig et al., 2010] Treibig, J., Hager, G., and Wellein, G. (2010). Likwid : A lightweight performance-oriented tool suite for x86 multicore environments. In *2010 39th International Conference on Parallel Processing Workshops*, pages 207–216.

[Weaver, 2013] Weaver, V. M. (2013). Linux perf_event features and overhead.

[Williams et al., 2009] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline : An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4) :65–76.

[Xiang et al., 2010] Xiang, L., Yuan, Q., Zhao, S., Chen, L., Zhang, X., Yang, Q., and Sun, J. (2010). Temporal recommendation on graphs via long- and short-term preference fusion. pages 723–732.

Annexe : Extraits de codes sources analysés

A. Codes pour GraFC2TC

A.1. Code pour la région de calcul de la matrice de similarité

```
//Compute implicit Rating matrix to be used for Jaccard similarity computation.
        LIKWID_MARKER_INIT;
        LIKWID_MARKER_THREADINIT;
        LIKWID_MARKER_START("jaccard");
        for(u_i=0; u_i<nb_user; u_i++){
            for(i_i=0; i_i<nb_item; i_i++){
                double rrr = rating_matrix[u_i][i_i];
                if((rrr >= m_rating_median) && (rrr >= user_rating_mean[u_i]))
                    implicit_rating_matrix[u_i][i_i] = 1.0;
                else
                    implicit_rating_matrix[u_i][i_i] = 0.0;
            }
        }
        // Compute user jaccard similarity matrix.
        double sim_u1_u2;
        for(int u_i_1=0; u_i_1<nb_user; u_i_1++){
            for(int u_i_2=u_i_1; u_i_2<nb_user; u_i_2++){
                sim_u1_u2 = jaccard_similarity_score(implicit_rating_matrix[u_i_1],
                                                     implicit_rating_matrix[u_i_2]);
                user_jaccard_similarity[u_i_1][u_i_2] = sim_u1_u2;
                user_jaccard_similarity[u_i_2][u_i_1] = sim_u1_u2;
            }
        }
        LIKWID_MARKER_STOP("jaccard");
        LIKWID_MARKER_CLOSE;
```

A.2. Code pour le Calcul de PageRank

```
//# power iteration: make up to max_iter iterations
    for (i = 0; i < max_iter; i++){
       std::vector<double>  tmp(N, 0.0);
       double sum = 0.0, temp;
       for(int l=0; l<N; l++){
            xlast[l] = x[l];
       }
       for (j = 0; j < dangling_size; j++){
            sum += x[is_dangling[j]];
       }
```

```
        // compute the x*M product for new x vector
        for (k = 0; k < size_product; k++){
             int ind_ligne = product.data[k]->row, ind_colone = product.data[k]->col;
             tmp[ind_colone] += x[ind_ligne] * product.data[k]->value;
        }
        for (k = 0; k < tmp.size(); k++){
             x[k] = alpha * (tmp[k]+(sum*dangling_weights[k])) + ((1.0-alpha)*p[k]);
        }
        double err = 0.0, tmp_er = 0.0;
        for (k = 0; k < N; k++){
             tmp_er = x[k] - xlast[k];
             err += std::abs(tmp_er);
        }
        if(err < N * tol){
             for (k = 0; k < N; k++){
                 result[nodes[k]->name] = x[k];
             }
             return result;
        }
    }
    for (k = 0; k < N; k++){
         result[nodes[k]->name] = x[k];
    }
    return result;
}
```

B. Codes pour PageRank

B.1. Code pour la lecture des données sur disque

```
  while(!feof(fp)){

    fscanf(fp,"%d%d",&fromnode,&tonode);
    if (fromnode > cur_row) { // change the row
      curel = curel + elrow;
      for (int k = cur_row + 1; k <= fromnode; k++) {
        row_ptr[k] = curel;
      }
      elrow = 0;
      cur_row = fromnode;
    }
    val[i] = 1.0;
    col_ind[i] = tonode;
    row_ind[i] = cur_row;
    elrow++;
    i++;
  }
  row_ptr[cur_row+1] = curel + elrow - 1;
```

B.2. Code pour la boucle de calcul du pageRank

```
while (looping){
    // Initialize p_new as a vector of n 0.0 cells
    for(i=0; i<n; i++){
      p_new[i] = 0.0;
    }
    int rowel = 0, curcol = 0;

    /* Page rank modified algorithm */

    LIKWID_MARKER_START("pageRank");
    //csr_spmv(n, row_ptr, col_ind, val, p, p_new);
    coo_spmv(e, row_ind, col_ind, val, p, p_new);
    LIKWID_MARKER_STOP("pageRank");

    // Adjustment to manage dangling elements
    for(i=0; i<n; i++){
      p_new[i] = d * p_new[i] + (1.0 - d) / n;
    }
    // TERMINATION: check if we have to stop
    float error = 0.0;
    for(i=0; i<n; i++) {
      error =  error + fabs(p_new[i] - p[i]);
    }
    //if two consecutive instances of pagerank vector are almost identical, stop
    if (error < 0.000001){
      looping = 0;
    }
    // Update p[]
    for (i=0; i<n;i++){
        p[i] = p_new[i];
    }
    // Increase the number of iterations
    k = k + 1;
}
```