THÈSE DE DOCTORAT DE
SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Florian Lemaitre

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Tracking haute fréquence pour architectures SIMD: optimisation de la reconstruction LHCb

présentée le 13 Février 2019

devant le jury composé de :

[Directeur de thèse]	Lionel	Lacassagne	LIP6	(Sorbonne Université)
[Rapporteur]	Albert	Cohen	Google	
[Rapporteur]	Daniel	Menard	IETR	(Université de Rennes)
[Examinateur]	Emmanuel	Chailloux	LIP6	(Sorbonne Université)
[Examinateur]	Michèle	Gouiffès	LIMSI	(Université Paris-Sud)
[Examinateur]	Bertrand	Le Gal	IMS	(Université de Bordeaux)

Remerciements

J'aimerais remercier toutes les personnes qui ont participé de près ou de loin à cette thèse :

Albert Cohen et Daniel Menard pour leurs précieux conseils et remarques rapportés suite à la lecture de mon manuscrit de thèse.

Emmanuel Chailloux, Michèle Gouiffès et Bertrand Le Gal pour avoir accepté de prendre de leur temps pour faire partie de mon jury de thèse.

Lionel Lacassagne pour son aide déterminante et son soutien durant ces 3 années de thèse, et ce malgré la distance et les complications administratives.

Benjamin Couturier, Sébastien Ponce, Marco Clemencic et Paul Seyfert avec qui j'ai beaucoup (trop) discuté de mes travaux au sein de l'équipe LHCb.

Luis Granado, Joao Barbosa et Flavio Pisani pour m'avoir supporté pendant 3 ans dans le même bureau.

Toute l'équipe LHCb pour m'avoir accueilli et permis de réaliser cette thèse dans un tel environnement.

Ma compagne Charlotte à qui je dois désormais de nombreuses nuits de sommeil.

Table des matières

1 Introduction **7**

1.1 Présentation du CERN et du LHC 7

1.2 Présentation de LHCb 9

1.3 LHCb *Upgrade* pour 2020 10

1.4 Enjeux de la thèse 12

1.5 Plan de la thèse 12

2 Optimisations **15**

2.1 Architecture 15

2.1.1 SIMD 15

2.1.2 Cache 28

2.1.3 Scalarisation 34

2.1.4 Déroulage de boucle 34

2.2 Précision des calculs 39

2.2.1 Rappels sur le calcul flottant 39

2.2.2 Racine carrée inverse 41

2.3 Génération de code 49

2.3.1 Déroulage total 49

2.3.2 Dérouler-entrelacer et scalarisation 50

2.3.3 SIMD 53

2.4 Synthèse 55

3 Études préliminaires : calcul de paraboles **57**

3.1 Présentation des algorithmes 57

3.2 Étude sur la mesure du temps 60
3.2.1 Utilitaires de mesure du temps 60
3.2.2 Distribution du temps de traitement 62
3.3 Analyse des performances 66
3.3.1 Comparaison des *memory layouts* 66
3.3.2 Influence de la taille des données 67
3.3.3 Impact de la précision 69
3.3.4 *Multithreading* 70
3.4 Analyse de la précision 73
3.5 synthèse 74

4 Factorisation de Cholesky 75
4.1 Algorithme 75
4.1.1 Factorisation 75
4.1.2 Résolution 76
4.2 Transformations 76
4.2.1 Traitement par lot 77
4.2.2 Déroulage et scalarisation 78
4.2.3 Racine carrée inverse 81
4.3 Analyse des résultats 81
4.3.1 Protocole 81
4.3.2 Performance des différentes fonctions 83
4.3.3 Influence de la taille des données 84
4.3.4 Influence des optimisations 86
4.3.5 Comparaison avec les *wrappers* SIMD 87
4.3.6 Comparaison avec la MKL 88
4.3.7 Influence du déroulage 89
4.3.8 Influence de la taille des matrices 90
4.3.9 Autres architectures 90
4.3.10 Passage à l'échelle : OpenMP 93
4.4 Synthèse 94

5 Filtre de Kalman **95**
5.1 Algorithme 95
5.2 Transformations 98
5.2.1 Optimisation des accès aux matrices symétriques 98
5.2.2 Transformations algébriques 98
5.2.3 Heuristique de réordonancement 100
5.3 Analyse des résultats 101
5.3.1 Protocole 101
5.3.2 Influence des optimisations 101
5.3.3 Performance totale 103
5.4 Filtre de Kalman spécifique à LHCb (LHC-Beauty) 106
5.4.1 Algorithme 106
5.4.2 Algorithmes alternatifs 108
5.4.3 Résultats 110
5.5 Synthèse 114

6 Conclusion **115**

Bibliographie **117**

Glossaire **127**

Chapitre 1

Introduction

1.1 Présentation du CERN et du LHC

Le CERN (Organisation Européenne pour la Recherche Nucléaire) est un pilier de la recherche fondamentale depuis sa création en 1954. Situé à la frontière Franco-Suisse et comptant 22 États membres, le CERN est devenu le plus gros laboratoire de physique des particules du monde. À présent, plus de 10 000 physiciens et ingénieurs travaillent ensemble à l'étude du monde sub-atomique.

Le CERN a aussi permis des avancées en dehors du monde de la physique avec par exemple la création du *World Wide Web* au début des années 90 : le premier site web fut activé en 1991, et le 30 Avril de la même année, le CERN annonce que le web sera gratuit et libre pour tout le monde.

Afin d'étudier les particules élémentaires (les composants de toute forme de matière), le CERN a mis au point l'appareil scientifique le plus grand et le plus complexe jamais réalisé : le LHC (*Large Hadron Collider*). Le Grand Collisionneur de Hadron est un accélérateur de particule circulaire de 27 km de circonférence enterré à presque 175 mètres sous le sol. Il a permis aux chercheurs du CERN et du monde entier de faire des découvertes majeures en physique des particules. La plus connue est la découverte de la dernière pièce manquante du Modèle Standard : le boson de Brout-Englert-Higgs.

Sur la figure 1.1, on peut voir tout le complexe du CERN avec les différents accélérateurs de particules.

Le LHC comporte 4 expériences principales : Alice (*A Large Ion Collider Experiment*), Atlas (*A Toroidal LHC Appartus*), CMS (*Compact Muon Solenoid*) et LHCb. Chaque expérience consiste en un point de collision entre les deux faisceaux de protons du LHC, un détecteur pour mesurer les particules générées par la collision, ainsi qu'une ferme de calculs de plusieurs milliers de machines pour filtrer et analyser les données en sortie du détecteur.

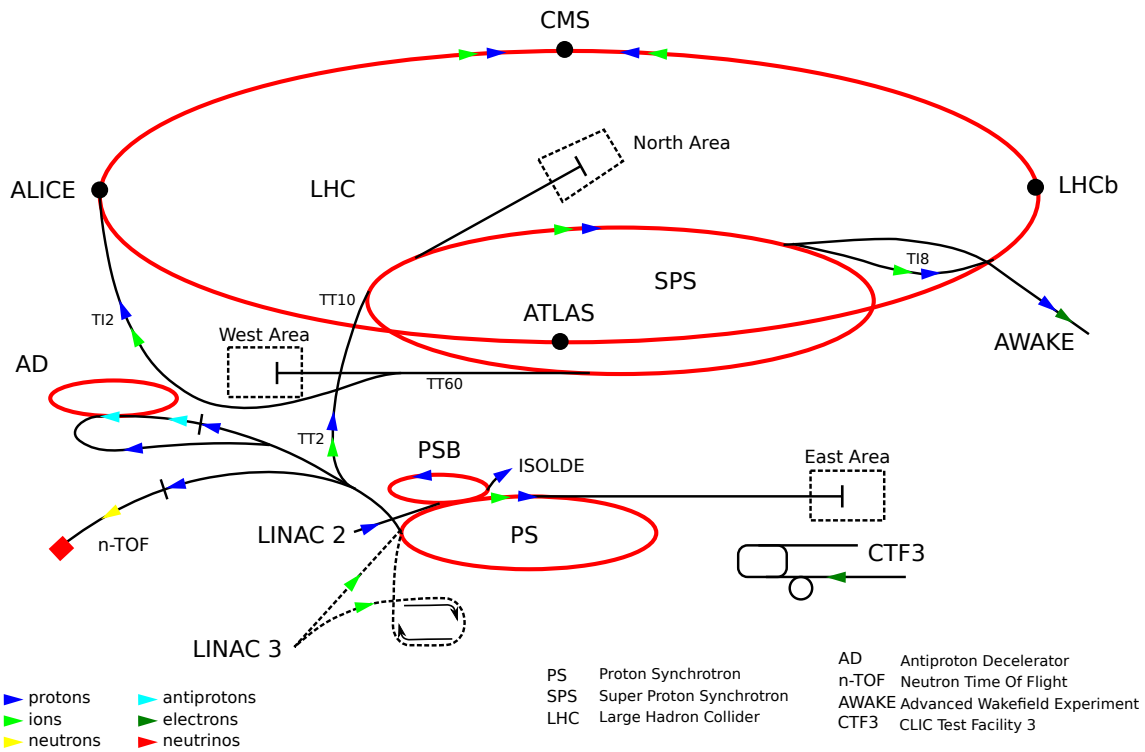


Figure 1.1 – Schéma des installations du CERN

1.2 Présentation de LHCb

LHCB est l'une des 4 expériences principales du LHC et est conçu pour explorer et analyser les différences entre la matière et l'anti-matière. Plus précisemment, le but est de répondre au problème de l'anti-matière manquante en analysant les propriétés fondamentales de l'anti-matière.

Le détecteur est composé d'un aimant tordant la trajectoire des particules chargées, ainsi que de plusieurs sous-détecteurs (figure 1.2) :

VeLo : Le *Vertex Locator* est un petit détecteur de particule extrêmement précis. Il est le plus proche possible du point de collision.

RICH1 : Le premier détecteur à effet Cherenkov (*Ring Imaging Cherenkov*) se situe juste après le VeLo et permet d'identifier et de mesurer la vitesse des particules lentes.

Tracker Turencis : Ce sous-détecteur est un *tracker* : il permet de déduire la position et la direction des particules le traversant. Celui-ci est situé juste avant l'aimant et possède 2 stations (plans de détections)

Tracker principal : Ce *tracker* est situé juste après l'aimant et possède 3 stations.

RICH2 : Ce deuxième détecteur à effet Cherenkov permet de mesurer la vitesse des particules rapides

Calorimètres : Les calorimètres, situés après le *tracker* principal et le RICH2, permettent de mesurer l'énergie des particules rapides. Ils permettent aussi de détecter les évènements potentiellement intéressants avant la reconstruction.

Chambres à muons : Les chambres à muons permettent de détecter les muons : des particules lourdes, n'intéragissant que peu avec la matière. Ces détecteurs permettent de classifier la nature de la collision avant la reconstruction.

L'expérience LHCb détecte 40 000 000 de collisions par seconde, lorsque le LHC est en route. Cela produit une quantité phénoménale de données qui ne peut pas être analysée ni même stockée aussi vite. Juste en sortie du détecteur se trouvent 350 FPGA formant le LLT (*Low Level Trigger*) qui permettent de filtrer les évènements pour ne conserver que les évènements potentiellement intéressants. Les données filtrées sont ensuite transmises au HLT (*High Level Trigger*) constitué de 2000 serveurs.

Le traitement du HLT est séparé en deux étapes (HLT1 et HLT2). La première étape permet de reconstruire les trajectoires des particules. Le deuxième étape rassemble toutes les données du HLT1 afin de reconstruire l'évènement dans sa globalité. Ces deux étapes agissent également comme un filtre et ne conservent que les évènements jugés potentiellement intéressants.

Les données sont mises dans un tampon avant d'être traitées. En considérant le nombre de collisions et les pauses entre les *batchs* de collisions (pour réinjecter des protons par exemple), il faut en moyenne qu'un évènement soit traité en 10 ms par cœur.

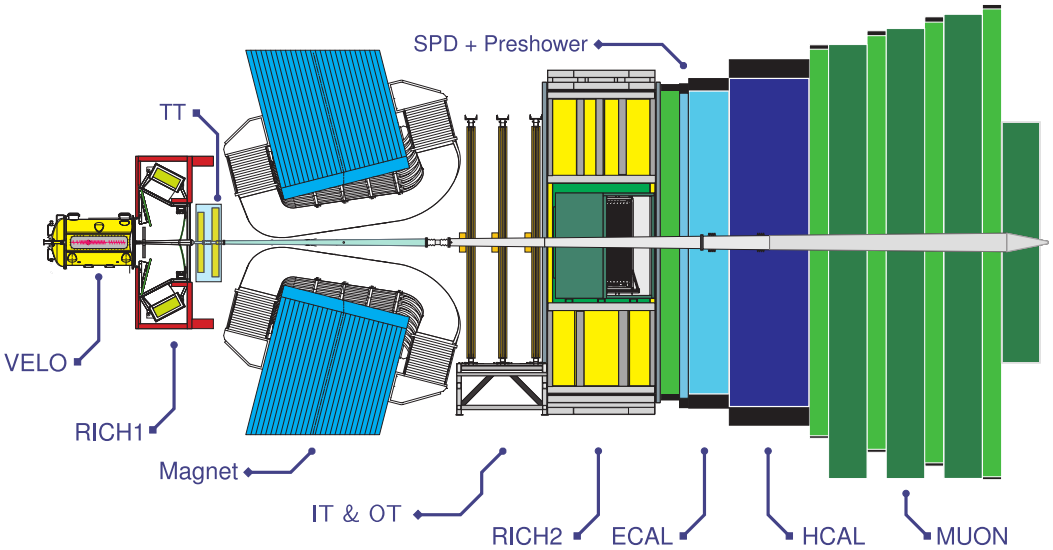


Figure 1.2 – Schéma du détecteur LHCb

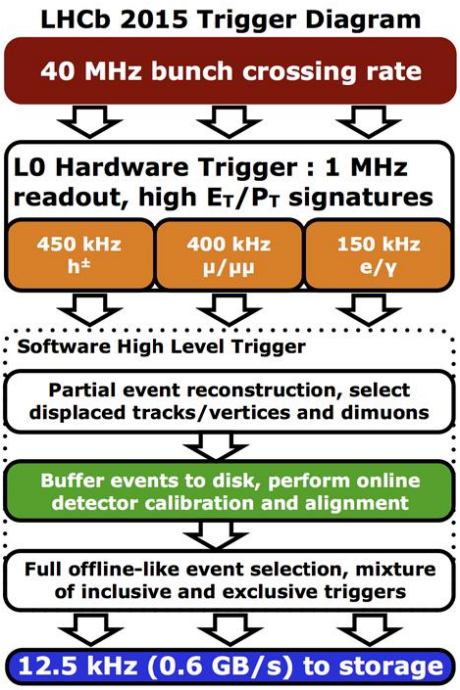
Les données sont ensuite envoyées à la surface où elles seront stockées sur bandes magnétiques et également envoyées sur un réseau de calculs mondial appelé la Grille. Les évènements envoyés sur la Grille sont analysés plus profondément et les résultats sont aggrégés sur plusieurs millions d'évènements.

Tout le code de reconstruction et d'analyse est écrit en C++ et utilise un *framework* interne appelé Gaudi.

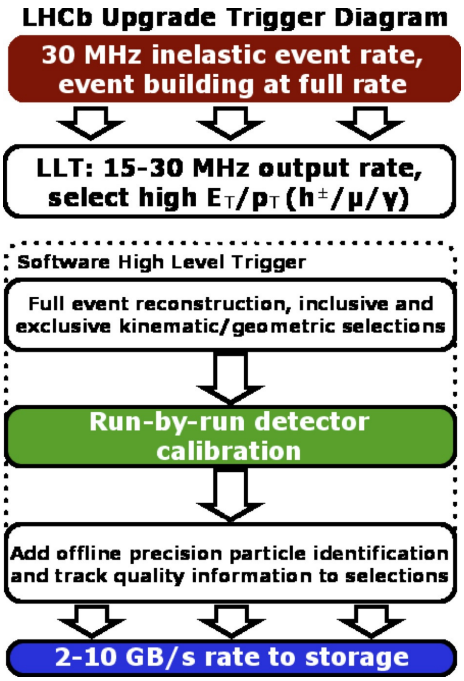
1.3 LHCb *Upgrade* pour 2020

Le détecteur LHCb va être amélioré pour 2020, date de la prochaine mise en route du LHC. Beaucoup de sous-détecteurs seront changés et leur architecture différente nécessite de réécrire une partie des algorithmes de reconstruction [1].

En outre, la chaine de traitement va également changer : le filtre matériel LLT disparaitra et sera intégré au HLT (figure 1.3). La conséquence directe est que le HLT1 dans sa globalité devra traiter 30 fois plus d'évènement à la seconde. L'achat de machines plus récentes permettra d'augmenter considérablement la puissance de calcul de la ferme. Cela ne sera toutefois pas suffisant pour arriver au facteur 30 requis. Les estimations indiquent qu'il faut accélérer le code de la reconstruction d'un facteur 6 [2]. C'est pourquoi il est crucial d'accélérer le code de reconstruction. Le filtre de Kalman permettant de filtrer les traces reconstruites représente presque la moitié du temps d'exécution de la séquence complète, et est donc un candidat privilégié pour l'accélération de la reconstruction.



(a) Reconstruction actuelle



(b) Reconstruction en 2020

Figure 1.3 – Schéma de la reconstruction des évènements au sein de LHCb

1.4 Enjeux de la thèse

Les algorithmes de la reconstruction LHCb sont *embarrasingly parallel*. En effet, la reconstruction d'un évènement est complètement indépendante de la reconstruction des autres évènements, et il y a 30 000 000 d'évènements à la seconde à traiter. De plus, chaque évènement nécessite le traitement de plusieurs centaines à plusieurs milliers de traces.

L'algèbre linéaire requise pour le traitement de ces traces est de faible dimension : typiquement de 2×2 à 5×5. Les bibliothèques d'algèbre linéaire telles que Eigen [3], Magma [4] ou MKL [5] sont optimisées pour de très grandes matrices, et ne sont pas donc adaptées à ces toutes petites matrices. Les petites matrices sont encore assez peu étudiées, mais leur grand nombre pose problème et nécessite des optimisations spécifiques. La communauté scientifique commence cependant à étudier des tels problèmes [6–10]. Les matrices de petites tailles ne sont pas utilisées qu'en physique des hautes énergies, mais aussi en vision par ordinateur [11, 12].

La latence des transferts de données entre CPU et GPU étant trop grande pour ce genre de problème lors du début de cette thèse, nous avons décidé de nous concentrer sur les architectures CPU SIMD.

Cette thèse a pour but d'améliorer la vitesse de traitements de problèmes à faible dimension dans un contexte *embarrasingly parallel*.

Ces contributions sont multiples : l'identification d'un manque d'optimisations en faible dimension dans l'État de l'Art, la mise au point d'un assemblage de transformations connues efficace en faible dimension, l'élaboration d'une méthode et d'un générateur de code portable permettant l'application de ces transformations, ainsi que des implémentations rapides pour les algorithmes de la factorisation de Cholesky et du filtre de Kalman.

1.5 Plan de la thèse

Nous verrons dans le chapitre 2 plusieurs transformations et optimisations relatives à l'architecture des processeurs telles que les agencements mémoires (*Array of Structures*, *Structure of Arrays*, *Array of Structures of Array*), la scalarisation et le déroulage de boucle (déroulage total, déroulage avec entrelacement). Nous ferons quelques rappels sur le calcul flottant, et comment accélérer la racine carrée inverse. Puis nous présenterons un générateur de code implémentant la plupart de ces transformations.

Dans le chapitre 3, nous utiliserons des algorithmes simples de la reconstruction LHCb afin de mettre au point une méthode pour mesurer de manière fiable les temps d'exécutions, d'appliquer et d'analyser les transformations vues précédemment sur ces algorithmes, et de faire une analyse de la précision des résulats.

Dans le chapitre 4, nous appliquerons la méthode ainsi que les transformations précédentes afin d'accélérer la factorisation de Cholesky pour de petites matrices. La factorisation de Cholesky est un algorithme classique d'algèbre linéaire qui est utilisé notamment pour inverser des matrices symétriques comme dans le filtre de Kalman.

Dans le chapitre 5, nous appliquerons une fois de plus ces transformations afin d'accélérer le filtre de Kalman, pièce maîtresse de la reconstruction LHCb. L'étude portera tout d'abord sur la formulation générale du filtre de Kalman avec des matrices 4×4, où de nouvelles transformations seront testées. Puis nous aborderons la formulation spécifique à LHCb du filtre de Kalman que nous comparerons en termes de vitesse à l'implémentation actuelle.

Chapitre 2

Optimisations

Cette partie traite des optimisations utilisées. Les optimisations présentées ici sont largement répandues dans la littérature [13]. Leurs combinaisons sont en revanche peu approfondies, mais sont importantes pour augmenter la vitesse de traitement [14]. Nous proposons ici des explications poussées de ces optimisations, ainsi que leurs implémentations au sein de cette thèse.

Le but de cette partie est de comprendre et d'assimiler plusieurs transformations et optimisations relatives à l'architecture des processeurs telles que les agencements mémoires (*Array of Structures*, *Structure of Arrays*, *Array of Structures of Array*), la scalarisation et le déroulage de boucle (déroulage total, déroulage avec entrelacement). Mais aussi l'accélération de la racine carrée inverse par des considérations arithmétiques. La plupart de ces transformations sont ensuites regroupées au sein d'un générateur de code qui sera utilisé dans la suite de cette thèse.

2.1 Architecture

2.1.1 SIMD

Afin d'accélérer les calculs, les processeurs actuels possèdent des unités fonctionnelles spécifiques qui sont capables d'effectuer une opération arithmétique comme une addition ou une multiplication sur des données différentes en parallèle. L'intérêt est le suivant : une telle instruction nécessite le même temps que la même instruction scalaire, mais traite plus de données dans le même laps de temps.

Ce paradigme est désigné par l'acronyme anglais SIMD de la classification Flynn [15] : *Single Instruction Multiple Data*. Les architectures SIMD les plus courantes sont SSE, AVX et AVX512 sur x86, Neon sur ARM ainsi que Altivec et VSX sur Motorola et IBM.

Cela se présente sous la forme d'un jeu de registres de taille fixes (typiquement 128 bits), et d'instructions spécifiques afin de manipuler ces registres. Les instructions disponibles se regroupent en plusieurs catégories usuelles comme les accès mémoires, les opérations arithmétiques et logiques ; mais aussi des catégories plus spécifiques au SIMD comme le mélange ou le masquage des éléments du vecteur.

Les instructions disponibles dépendent fortement de l'architecture ciblée. Il est donc nécessaire d'avoir une connaissance approfondie de l'architecture afin d'utiliser pleinement les unités SIMD.

Il existe plusieurs manières d'utiliser le SIMD :

— Assembleur
— *Intrinsics* : extension du langage C permettant un accès direct aux registres SIMD ainsi qu'aux instructions associées par le biais de types natifs et de fonctions.
— Bibliothèques SIMD : fonctionne comme les *intrinsics*, mais avec une interface abstraite identique entre les architectures : Boost.SIMD [16], Cyme [17], libsimdpp [18], MIPP [19], UME::SIMD [20], Vc [21], Vcl [22]...
— Paradigmes de programmation parallèle [23] : ISPC [24], OpenACC [25], OpenCL [26], OpenMP [27], `#pragma`s...
— Vectorisation par le compilateur : ce dernier abstrait les opérations arithmétiques scalaires afin de les regrouper en opérations parallèles qu'il pourra alors convertir en instructions SIMD.

L'assembleur ainsi que les *intrinsics* sont particulièrement difficiles à écrire et à maintenir, et n'offrent aucune portabilité. Chaque architecture possède ses propres instructions et ses propres *intrinsics*, même si les capacités sont similaires.

Les bibliothèques SIMD avec leur abstraction permettent de s'affranchir des problèmes de portabilité, mais restent difficiles à programmer et requièrent tout de même une bonne compréhension de l'architecture afin d'avoir de bonnes performances.

Les paradigmes de programmation parallèle ainsi que la vectorisation offrent une portabilité maximale à moindre effort, mais perdent le contrôle requis pour des performances optimales.

Les accès mémoires en SIMD doivent être alignés avec la taille du registre. Par exemple, si un registre SIMD fait 16 octets, il faut que l'adresse soit un multiple de 16. Cette contrainte est imposée par l'implémentation matérielle afin d'être efficace.

Sur la plupart des architectures, il existe également des instructions pouvant faire des accès non-alignés, mais celles-ci sont moins efficaces : elles sont souvent implémentées avec deux accès alignés. Les architectures x86 supportent les accès non-alignés aussi rapides que les accès alignés depuis Nehalem chez Intel et depuis Bulldozer chez AMD.

C'est pourquoi il est préférable d'aligner les données en mémoire afin de pouvoir utiliser les accès alignés plus efficaces. Pour ceci, il est nécessaire de recourir à des fonctions d'allocations mémoires spécifiques comme `posix_memalign` [28, pp. 1448–1449], `_mm_malloc` [29, pp. 657] ou bien `aligned_alloc` [30, chapitre 7.22.3.1] en C11.

2.1.1.1 Historique

Les premiers travaux sur des architectures vectorielles remontent au début des années 60 avec le projet "Solomon" de Westinghouse. Ce projet fut rapidement arrêté, mais servit de base pour le projet ILLIAC IV [31]. Sorti en avril 1972, ILLIAC IV est ainsi la première machine vectorielle et comptait 64 FPU (*Floating Point Unit*) à 64 bits. Toutes les FPU sont indépendantes et possèdent leur propre mémoire, mais exécutent la même instruction. Cette approche fait de l'ILLIAC IV la première machine SIMD. Elle n'est cependant stable qu'à partir de 1975.

Entre-temps, deux autres machines vectorielles apparurent en 1974 : le ASC [32] (Advanced Scientific Computer) de Texas Instruments, et le STAR-100 [33] de CDC (Control Data Corporation). Sur ces machines, une instruction vectorielle "crée" un pipeline alimentant des unités scalaires. Les données sont lues et écrites en mémoire directement. Du fait que les éléments d'un vecteur ne sont pas calculés en parallèle, ces machines ne sont pas considérées comme des machines SIMD, mais des machines vectorielles.

Ces premières machines seront vite détrônées par les machines Cray-1 en 1975 [34]. Cette architecture a une approche vectorielle similaire à l'ASC et au STAR-100, mais ajoute des registres vectoriels. Ces registres permettent de stocker les résultats temporaires des opérations vectorielles, ce qui permet de chaîner les instructions et ainsi commencer l'instruction suivante alors même que la précédente n'a pas terminé de traiter le vecteur complet. Cette approche permet aussi de s'affranchir des accès mémoires entre deux instructions travaillant sur les mêmes données. En revanche, ces registres imposent une taille maximale aux vecteurs traités (64 éléments de 64 bits).

Les machines Cray sont des machines phares des années 80 et beaucoup portent une grande estime pour ces machines aujourd'hui encore. Par la suite, les autres fabricants utiliseront également une approche vectorielle pour leurs supercalculateurs.

Cette période voit également l'arrivée de machines SIMD massivement parallèles possédant plusieurs milliers de petites ALU (*Arithmetic and Logic Unit*) indépendantes nommées PE (*Processing Element*). Le DAP [35] (Distributed Array Processor) d'ICL (International Computers Limited) en est le précurseur avec un premier prototype en 1974 et une sortie officielle en 1979. Il possède 4096 PE de 1 bit.

Plusieurs constructeurs suivent cette voie jusqu'aux années 90. Le représentant le plus connu de ce type de machines est la "Connection Machine" [36] sortie en 1985 avec ses 65 536 PE de 1 bit, connectés selon un hypercube.

L'intérêt pour les architectures vectorielles et SIMD dédiées aux supercalculateurs s'essoufle dans les années 90 au profit d'architectures multi-cœurs et surtout multi-processeurs. Par la suite, les supercalculateurs, bénéficiant des avancées des microprocesseurs, disposent de nouvelles architectures SIMD.

Du côté des microprocesseurs, un intérêt pour le SIMD commence à apparaître au milieu des années 90 afin d'accélérer les applications multimedia (comme le décodage vidéo). Par simplicité, les architectures existantes sont étendues pour supporter de nouvelles instructions SIMD. L'approche utilisée est de stocker un vecteur en registre (comme les machines Cray), et d'utiliser autant d'unités fonctionnelles que nécessaire pour traiter tous les éléments en parallèle. On parle ainsi de SWAR [37] (*SIMD Within A Register*). Selon les opérations, il est possible d'utiliser des unités fonctionnelles plus grosses et d'en modifier légèrement le circuit : ainsi, deux additionneurs 16 bits peuvent être implémentés en utilisant un additionneur 32 bits où la retenue du 16ème bit n'est pas propagée au 17ème bit, séparant ainsi le traitement en deux sans nécessiter d'unités fonctionnelles spécifiques. On parle dans ce cas de *Sub-Word Parallelism*.

La première architecture à se voir doter d'une telle extension SIMD est PA-RISC avec les jeux d'instructions MAX-1 en 1994, puis MAX-2 en 1995, d'une largeur respective de 32 et 64 bits. Les autres architectures suivent avec VIS pour SPARC en 1995, MDMX pour MIPS-V en 1996, MVI pour Alpha en 1996 et MMX pour x86 en 1997, toutes avec une largeur de 64 bits. Ces extensions étant orientées multimedia, elles ne supportent que des données de type entier. La plupart de ces jeux d'instructions n'introduisent pas de nouveaux registres et réutilisent des registres existants : par exemple, MMX utilise les registres flottants du coprocesseur x87, ce qui pose problème lors de l'utilisation simultanée d'instructions scalaires et SIMD.

Sur MIPS-V, le jeu d'instructions "paired-single" sorti en 1994 permet d'étendre le SIMD au calcul flottant simple précision. AMD développe 3DNow ! en 1998 qui permet lui aussi le support des nombres flottants simple précision sur x86. 3DNow ! est une extension de MMX.

En 1998, Apple, IBM, et Motorola développent conjointement un jeu d'instructions SIMD 128 bits pour l'architecture Power. Ce jeu d'instructions est appelé "Velocity Engine" par Apple, VMX (*Vector Media Extension*) par IBM et enfin Altivec par Motorola. Le nom Altivec est désormais préféré, et le nom "Velocity Engine" a complètement disparu. Ce jeu d'instructions supporte l'arithmétique flottante simple précision, et peut donc traiter jusqu'à 4 flottants en parallèle. Contrairement à ses prédecesseurs, Altivec introduit 32 registres de 128 bits. Il est utilisé pour accélérer les applications Apple comme QuickTime ou iTunes, mais aussi des applications tierces telles qu'Adobe Photoshop. Il est considéré, aujourd'hui encore, comme l'un des meilleurs jeux d'instructions SIMD de par sa souplesse.

En 1999, Intel présente SSE, un jeu d'instructions SIMD 128 bits. Il introduit 8 registres de 128 bits (`XMM0`–`XMM7`). SSE ne supporte pas les types entiers et est donc restreint au calcul flottant simple précision. Il faut donc utiliser MMX afin d'utiliser du SIMD avec des entiers.

ARM spécifie un premier jeu d'instructions vectoriel en 2001 : VFP (*Vector Floating-Point*). Ce jeu d'instructions définit 16 registres 64 bits pouvant également être interprétés comme 32 registres 32 bits. Plusieurs registres consécutifs peuvent être combinés pour former un vecteur de taille maximale 256 bits. À l'instar des machines Cray, les instructions vectorielles ne traitent pas les éléments du vecteur en parallèle, mais séquentiellement au travers d'un pipeline. Ce jeu d'instructions ne supporte que les opérations flottantes (en simple et double précision).

L'année suivante, ARM spécifie un jeu d'instructions SIMD afin d'accélérer les applications multimedias. Ce dernier n'a pas de nom officiel. Il est souvent désigné par "*SIMD extensions for multimedia*". Tout comme les premières architectures SWAR, il n'introduit aucun registre et permet d'effectuer des opérations entières sur 2×16 bits ou 4×8 bits (largeur totale : 32 bits).

Au début des années 2000, les premières cartes graphiques programmables apparaissent avec une architecture similaire à ILLIAC IV.

En 2001, une nouvelle version de SSE voit le jour. SSE2 ajoute le support des flottants double précision, des entiers 8, 16 et 32 bits, ainsi qu'un support partiel des entiers 64 bits. Les opérations sur les entiers sont inspirées des opérations existantes sur MMX. SSE comme SSE2 supportent des instructions scalaires sur les registres vectoriels (qui ne traitent donc que le premier élément). Cela permet de s'affranchir du coprocesseur x87 pour le calcul flottant, et ainsi avoir un support des flottants selon la norme IEEE 754 [38]. Dès lors, les compilateurs n'utilisent plus x87 pour compiler du code en `float` ou en `double` lorsque SSE2 est disponible, et ce même pour du code scalaire non-vectorisé. Le coprocesseur x87 reste utilisé si le code utilise du `long double` (précision étendue).

Plusieurs versions de SSE se suivront avec des légères améliorations jusqu'en 2008 : SSE3 (2004), SSSE3 (2006), SSE4.1 (2007), SSE4a (2007) et SSE4.2 (2008).

En 2008, le jeu d'instructions AVX est spécifié par Intel et AMD. Il faudra attendre 2011 pour que les premiers processeurs le supportent. L'apport principal de l'AVX est une largeur de 256 bits. AVX n'introduit pas de nouveau registre, mais augmente la taille des registres SSE. En outre, le nouvel encodage VEX des instructions autorise trois opérandes par instructions, ce qui peut réduire la pression de registres.

Un support amélioré des entiers (notamment 64 bits) et de nouvelles instructions de mélange voient le jour en 2013 avec la version suivante : AVX2. Le support des FMA (*Fused Multiply-Add*) apparaitra en même temps.

En 2008, ARM définit un nouveau jeu d'instructions SIMD remplaçant l'ancien : Neon (ou "Advanced SIMD"). Neon définit 16 registres 128 bits. Ces registres peuvent être accédés également comme 32 registres 64 bits, et sont partagés avec la vfp. Neon supporte les entiers 8, 16, 32 et 64 bits, les flottants simple précision, mais également les polynômes booléens de degrés 8 et 16.

En 2009, le jeu d'instructions VSX complémente Altivec. VSX apporte le support des entiers 64 bits, de la double précision, ainsi que la division et la racine carrée précise.

En 2012, IBM créé le jeu d'instructions SIMD QPX de 256 bits. Ce jeu d'instructions est prévu pour le calcul scientifique et ne supporte que les flottants double précision.

En 2013, Intel propose une version étendue à 512 bits d'AVX : AVX512. Mise à part des registres plus larges, AVX512 apporte plusieurs nouveautés :

— un nombre accru de registres (32 registres de 512 bits),
— les instructions sont masquables : le traitement d'un élément en particulier peut être ignoré par le masque,
— 8 registres spécifiques aux masques,
— d'avantages d'instructions spécialisées comme la détection de conflit qui permet de supporter plus d'algorithmes.

AVX512 est une version améliorée du jeu d'instructions des Xeon Phi Knights Corner (déjà en 512 bits).

Fujitsu et ARM annoncent en 2017 un nouveau jeu d'instructions : SVE [39] (*Scalable Vector Extension*). SVE se démarque des autres architectures SWAR par une largeur de registre non fixée. En effet, chaque machine SVE peut implémenter une taille de registre différente. Le code assembleur pour cette architecture est donc agnostique à la taille des vecteurs. Tout comme AVX512, les instructions sont masquables.

Début 2018, NEC propose des machines vectorielles sous la forme de cartes accélératrices avec le SX-Aurora TSUBASA [40]. Cette machine adopte une approche hybride entre le SIMD, et le vectoriel (l'exécution séquentielle via un pipeline à la Cray). Les registres vectoriels ont une largeur de 16 384 bits (256×64). Ces registres sont découpés en tranches de 32 éléments : tous les éléments d'une même tranche sont traités en parallèle, et chaque tranche est traitée séquentiellement au sein d'un pipeline spécifique.

Le tableau 2.1 retrace l'évolution des processeurs vectoriels et SIMD depuis ILLIAC IV jusqu'au SX-Aurora TSUBASA de NEC.

Table 2.1 – Évolution des architectures CPU vectorielles et SIMD

Machine	Année	Type	Largeur
ILLIAC IV	1972	SIMD	64×64
CDC STAR-100	1974	vectoriel	N/A
TI ASC	1974	vectoriel	N/A
Cray-1	1975	vectoriel	64×64
ICL DAP	1979	SIMD	4096×1
Goodyear MPP	1983	SIMD	16 384×1
Cray-2	1985	vectoriel	64×64
NEC SX-2	1985	vectoriel	256×64
Connection Machine 1	1985	SIMD	65 536×1
CDC ETA10 [41]	1987	vectoriel	?
NEC SX-3 [42]	1990	vectoriel	256×64
Connection Machine 5 [43]	1991	MIMD	4×64

Architecture	extension	Année	Type	Largeur	I8	I16	I32	I64	F32	F64
PA-RISC	MAX-1	1994	SIMD	32	✓	✓				
PA-RISC	MAX-2	1995	SIMD	64	✓	✓	✓			
SPARC	VIS	1995	SIMD	64	✓	✓	✓			
MIPS-V	MDMX	1996	SIMD	64	✓	✓	✓			
MIPS-V	paired-single	1996	SIMD	64					✓	
Alpha	MVI	1996	SIMD	64	✓	✓	✓			
x86	MMX	1997	SIMD	64	✓	✓	✓			
x86	3DNow !	1998	SIMD	64					✓	
PowerPC	Altivec	1998	SIMD	128	✓	✓	✓		✓	
x86	SSE	1999	SIMD	128					✓	
ARM	VFP	2001	vectoriel	32–256					✓	✓
x86	SSE2	2001	SIMD	128	✓	✓	✓	∼	✓	✓
ARM	media extension	2002	SIMD	32	✓	✓				
x86	SSE3	2004	SIMD	128	✓	✓	✓	∼	✓	✓
x86	SSSE3	2006	SIMD	128	✓	✓	✓	∼	✓	✓
x86	SSE4.1	2007	SIMD	128	✓	✓	✓	∼	✓	✓
x86	SSE4.2	2008	SIMD	128	✓	✓	✓	✓	✓	✓
ARM	Neon	2008	SIMD	128	✓	✓	✓	∼	✓	
PowerPC	VSX	2009	SIMD	128	✓	✓	✓	✓	✓	✓
x86	AVX	2011	SIMD	256	✓	✓	✓		✓	✓
Power	QPX	2012	SIMD	256						✓
x86	AVX2	2013	SIMD	256	✓	✓	✓	✓	✓	✓
ARM	Neon (64 bits)	2013	SIMD	128	✓	✓	✓	✓	✓	✓
x86	AVX512	2017	SIMD	512	✓	✓	✓	✓	✓	✓
ARM	SVE	2017*	SIMD	128–2048	✓	✓	✓	✓	✓	✓
NEC	TSUBASA	2018	hybride	16 384	?	?	?	?	✓	✓

* : Date d'annonce

2.1.1.2 Métriques

De par sa nature, le SIMD nécessite un plus gros débit des bus et des mémoires. En effet, ces instructions traitent plus de données dans le même laps de temps que les instructions scalaires équivalentes. Il faut donc plus de données afin de garder les unités fonctionnelles occupées.

On mesure généralement la performance brute d'une machine en comptant le nombre d'opérations flottantes que celle-ci peut effectuer en une seconde : flops. On utilise généralement le préfixe giga (10^9) pour des ordres de grandeurs plus appréciables : Gflops. On distingue deux types de performance brutes : la performance théorique déduite des spécifications du processeur, et la performance pic soutenue mesurée grâce à des *benchmarks* normalisés dont le plus connu est Linpack [44]. La performance pic se situe généralement entre 80 et 90 % de la performance théorique, et reflète mieux ce qu'il est possible de tirer de la machine.

En ce qui concerne le débit des accès mémoires, on parle de bande passante qui s'exprime en octets par seconde (o/s). A l'instar de la performance, le préfixe giga est très souvent utilisé : Go/s.

Contrairement à la performance brute, il existe une hiérarchie mémoire avec plusieurs caches, chacun ayant une bande passante différente. Les *benchmarks* les plus utilisés normalisent les motifs d'accès mémoires. Le plus connu est STREAM TRIAD [45] (listing 2.1). Il est possible de mesurer la bande passante de la mémoire externe (RAM), des différents niveaux de caches ou bien entre deux processeurs. Nous nous intéresseront principalement à la bande passante des caches et de la mémoire externe. La bande passante théorique n'est que très rarement considérée dans ce contexte.

Il peut être intéressant de regarder le rapport entre les deux métriques précédentes. On peut ainsi définir le rapport calcul/bande passante ainsi : Le produit de la performance brute (pic ou théorique) par la taille d'un élément (4 octet/élément pour un `float`) divisé par la bande passante.

Le point intéressant d'une telle métrique est qu'elle ne dépend pas de la taille des

Listing 2.1 Implémentation naïve de STREAM TRIAD

```
void stream_triad(float* A, float* B, float* C, long n) {
  float alpha = 1.2345f;
  for (long i = 0; i < n; i++) {
    C[i] = alpha * A[i] + B[i];
  }
}
```

éléments dès lors que l'architecture supporte le SIMD. En effet, la très grande majorité des instructions SIMD prend le même nombre de cycles pour les calculs en simple précision qu'en double (la division et la racine carrée en sont des contre-exemples). Si l'instruction d'addition en `float` prend 3 cycles pour calculer 4 éléments, alors l'instruction d'addition en `double` prendra 3 cycles pour calculer 2 éléments. En d'autres termes, 4 additions en `float` nécessiteront 3 cycles, alors que 4 additions en `double` nécessiteront 6 cycles.

Cette métrique est à mettre en parallèle avec l'Intensité Arithmétique (IA) qui est le rapport entre le nombre d'opérations arithmétiques et le nombre d'accès mémoire d'une fonction ou d'un noyau de calcul. Ainsi, si l'IA est largement supérieure au rapport calcul/bande passante, alors le problème sera généralement limité par la puissance de calcul (*compute bound*). Dans le cas opposé, le problème sera généralement limité par les accès mémoires (*memory bound*). Quand les deux rapports sont proches, le programme aura, en pratique, de grandes chances d'être limité par d'autres facteurs comme le *sparse addressing* ou plus rarement le décodage des instructions par le processeur.

Comme la bande passante dépend de la mémoire qui est accédée, il en est de même pour le rapport calcul/bande passante. Ainsi, il est possible que le problème soit *compute bound* lorsque les données sont dans le cache L1, tout en étant *memory bound* lorsque les données sont en mémoire externe. Ce rapport dépend aussi, bien évidemment, de la machine considérée.

2.1.1.3 *Array of Structures*

Considérons un ensemble de points dans l'espace (3D) dont on souhaite calculer le barycentre. La manière la plus simple de représenter ceci en mémoire est d'avoir un tableau de quadruplets (listing 2.2). Ainsi, pour calculer le barycentre, il suffit d'itérer sur l'ensemble des points, et de calculer la moyenne de chaque coordonnée : listing 2.3. Cet agencement mémoire s'appelle *AoS* (*Array of Structures*) signifiant littéralement tableau de structures.

Un tel code n'est pas trivialement vectorisable. En effet, bien que notre type `Point` soit un quadruplet de `float`s qui rentre donc parfaitement dans un registre SIMD de 128 bits, le rôle de chaque composante n'est pas identique. Le poids est différent et nécessite un

Listing 2.2 Tableau de points (*AoS*)

```
struct Point {
  float x, y, z; // coordonées
  float w; // masse
};
Point points[N];
```

Listing 2.3 Calcul de barycentre : scalaire (*AoS*)

```
Point center = {0.f, 0.f, 0.f, 0.f};
for (int i = 0; i < N; i++) {
  center.x += points[i].w * points[i].x;
  center.y += points[i].w * points[i].y;
  center.z += points[i].w * points[i].z;
  center.w += points[i].w;
}
center.x /= center.w;
center.y /= center.w;
center.z /= center.w;
```

traitement particulier. À noter que la division finale peut être effectuée en scalaire afin de faciliter la vectorisation sans que cela n'impacte les performances d'une telle fonctions (si `N` est suffisamment grand).

Pour résoudre ce problème, il existe principalement deux approches :

— vectorisation verticale : chaque vecteur correspond à même un point (ou une partie d'un point pour des problèmes plus gros),

— vectorisation horizontale : chaque vecteur correspond à une même composante de points différents.

La vectorisation verticale est généralement privilégiée lorsque les composantes d'un problème sont nombreuses et partagent le même rôle : vecteur ou matrice de grande dimension. Mais celle-ci ne passe pas bien à l'échelle lorsque la taille des objets manipulés est petite, d'autant plus, si la taille de ces objets n'est pas une puissance de 2. De plus, les opérations à effectuer sur ces vecteurs peuvent être difficiles à exprimer lorsque les composantes ne partagent pas le même rôle.

La vectorisation horizontale résout ces problèmes, mais a un coût lors de la lecture/écriture des données. Il faut en effet transposer les données afin que chaque registre traite la même composante de plusieurs points. De plus, la vectorisation horizontale requiert généralement plus de registres actifs : un registre par composante, là où la vectorisation verticale traite plusieurs composantes avec un seul registre.

Dans le cas présent, les deux approches donnent des résultats légèrement différents en pratique : les sommes ne sont pas calculées dans le même ordre.

Toutefois, la vectorisation horizontale ne peut s'appliquer que si le nombre d'itérations de la boucle est un multiple du cardinal d'un registre SIMD (4 `float`s dans le cas présent). Dans le cas contraire, il ne sera pas possible d'utiliser la version SIMD pour l'intégralité de la boucle. Il faudra en effet traiter la fin de la boucle (qui ne tiendrait pas dans un

Listing 2.4 Calcul de barycentre : vectorisation verticale (*AoS*)

```
typedef float __attribute__((__vector_size__(16))) vec4f;
typedef int   __attribute__((__vector_size__(16))) vec4i;

Point center_of_mass(const Point* pts, int n) {
  vec4f center;
  vec4i mask = {true, true, true, false};
  vec4f ones = {1.f, 1.f, 1.f, 1.f};
  for (int i = 0; i < n; i++) {
    vec4f p = {pts[i].x, pts[i].y, pts[i].z, pts[i].w};
    vec4f wheight = {p[3], p[3], p[3], p[3]};
    center += (mask ? weight : ones) * p;
  }
  vec4f wheight = {center[3], center[3], center[3], center[3]};
  center /= mask ? wheight : ones;
  return {center[0], center[1], center[2], center[3]};
}
```

registre SIMD entier) de manière différente : soit en repassant en scalaire, soit en utilisant des masques SIMD pour ne traiter que les éléments nécessaires.

De plus, dans cet exemple, le code applique une réduction qui nécessite un traitement particulier non-nécessaire en scalaire.

2.1.1.4 *Structure of Arrays*

En pratique, il est possible de s'affranchir du coût de la transposition tout en conservant les avantages de la vectorisation horizontale [46]. Il suffit d'avoir en mémoire les données prêtes à être exploitées. Au lieu de stocker un tableau de quadruplets, on stocke quatre tableaux de scalaires (listing 2.6). Ainsi, une composante aura un adressage contigu en mémoire pour l'ensemble des points. Tous les `xs` sont à la suite des autres et peuvent être chargés en registre trivialement. Cet agencement mémoire se nomme *SoA* (*Structue of Arrays*) et est l'agencement par défaut en Fortran 77.

Il faut adapter légèrement le code scalaire afin de bénéficier de cet agencement mémoire (listing 2.7). La vectorisation horizontale de ce code est désormais triviale et peut être laissée au compilateur. Il faudra cependant l'autoriser à effectuer des optimisations sur les nombres à virgule flottante pouvant changer les résultats (à cause de la réduction flottante).

Avec un tel agencement mémoire, le corps de boucle SIMD est écrit de manière identique au corps de boucle scalaire : toute la complexité du SIMD semble avoir disparue. Les

Listing 2.5 Calcul de barycentre : vectorisation horizontale (*AoS*)

```
typedef float __attribute__((__vector_size__(16))) vec4f;

Point center_of_mass(const Point* pts, int n) {
  Vec4f sX = {0, 0, 0, 0};
  Vec4f sY = {0, 0, 0, 0};
  Vec4f sZ = {0, 0, 0, 0};
  Vec4f sW = {0, 0, 0, 0};
  for (int i = 0; i < n - 4; i += 4) {
    // transposition des éléments
    Vec4f x = {pts[i].x, pts[i+1].x, pts[i+2].x, pts[i+3].x};
    Vec4f y = {pts[i].y, pts[i+1].y, pts[i+2].y, pts[i+3].y};
    Vec4f z = {pts[i].z, pts[i+1].z, pts[i+2].z, pts[i+3].z};
    Vec4f w = {pts[i].w, pts[i+1].w, pts[i+2].w, pts[i+3].w};

    // calcul identique au cas scalaire
    sX += w*x;
    sY += w*y;
    sZ += w*z;
    sW += w;
  }

  // réduction (vecteurs -> scalaires)
  Point center;
  center.x = sX[0] + sX[1] + sX[2] + sX[3];
  center.y = sY[0] + sY[1] + sY[2] + sY[3];
  center.z = sZ[0] + sZ[1] + sZ[2] + sZ[3];
  center.w = sW[0] + sW[1] + sW[2] + sW[3];

  // reste
  for (int i = (n/4)*4; i < n; i++) {
    center.x += pts[i].w * pts[i].x;
    center.y += pts[i].w * pts[i].y;
    center.z += pts[i].w * pts[i].z;
    center.w += pts[i].w;
  }
  center.x /= center.w;
  center.y /= center.w;
  center.z /= center.w;
  return center;
}
```

Listing 2.6 Tableau de points (*SoA*)

```
struct Points {
  float *x, *y, *z;
  float *w;
};
```

Listing 2.7 Calcul de barycentre : scalaire (*SoA*)

```
Point center_of_mass(Points points, int n) {
  Point center = {0, 0, 0, 0};
  for (int i = 0; i < n; i++) {
    center.x += points.w[i] * points.x[i];
    center.y += points.w[i] * points.y[i];
    center.z += points.w[i] * points.z[i];
    center.w += points.w[i];
  }
  center.x /= center.w;
  center.y /= center.w;
  center.z /= center.w;
  return center;
}
```

performances, au contraire, sont bien meilleures qu'en scalaire ou en *AoS*. Les unités SIMD sont utilisées plus efficacement.

Si l'on compare les vitesses de traitement des différentes approches sur un processeur Intel Xeon E5-2683 v3 (tableau 2.2), on observe des performances différentes. La vectorisation verticale peut donner une version plus lente que la version scalaire si elle est mal implémentée. La vectorisation verticale n'est pas adaptée à ce problème avec une accélération maximale de $\times 1.6$. La vectorisation horizontale en *AoS* permet d'atteindre une accélération de $\times 2.0$. À noter que le compilateur applique la vectorisation horizontale ici.

Dès lors que l'on passe en *SoA*, les performances augmentent avec une accélération de $\times 3.99$. Cette accélération est très proche de l'optimale pour du SSE ($\times 4$ en simple précision).

2.1.2 Cache

Une autre partie importante des processeurs est le cache [47]. Le cache permet d'accélérer les accès mémoires, lorsque ceux-ci sont réguliers. Son rôle devient crucial avec le SIMD en faisant persister les données au plus près du processeur, permettant ainsi d'avoir des débits soutenus élevés.

L'idée est la suivante : si un élément en mémoire est chargé, le cache va alors stocker cet élément, ainsi que des éléments spatialement proches (localité spatiale). Les éléments sont groupés en blocs appelés lignes de cache. Lorsqu'un des ces éléments est accédé à nouveau, il n'est pas nécessaire de lire la mémoire externe, et l'élément peut être lu directement depuis le cache qui possède un débit plus important et une latence plus faible. Si un élément en mémoire externe est accédé et qu'il n'y a plus d'emplacement libre pour cet élément, il faut alors vider une ligne de cache pour mettre le nouvel élément. La ligne de cache éliminée est généralement une ligne qui n'a pas été utilisée récemment. On garde ainsi les éléments accédés fréquemment en cache (localité temporelle).

Table 2.2 – Vitesse de traitement du calcul de barycentre en SSE4 simple précision (Intel Xeon E5-2683 v3)

version	*AoS*	*SoA*
scalaire	2.67	2.67
vectorisée	1.34	0.67
SIMD vertical naïf	3.17	N/A
SIMD vertical	1.67	N/A
SIMD horizontal	1.34	0.67

Vitesses de traitement en cycle/point

Les contraintes matérielles empêchent un cache d'être à la fois gros et rapide. C'est pourquoi les architectures actuelles utilisent une hiérarchie de caches permettant d'avoir des caches extrêmement rapides, tout en ayant également des caches de grosse tailles.

Voici une configuration de caches typique à 3 niveaux :
— L1i : entre 32 Ko et 64 Ko pour les instructions (le programme),
— L1d : entre 32 Ko et 64 Ko pour les données,
— L2 : entre 256 Ko et 1 Mo,
— L3 : entre 1 Mo et 4 Mo par cœur, partagé entre les cœurs.
Les processeurs avec beaucoup de cœurs peuvent ainsi avoir un cache L3 de plus de 64 Mo.

2.1.2.1 Sortie de cache : taille des données

La mémoire externe et les différents caches ayant des bandes passantes différentes, la performance d'un programme dépend de la localisation des données utilisées. Si celles-ci sont dans le L1, le programme pourra accéder aux données beaucoup plus vite que si les données sont dans le L3 (voire en mémoire externe). C'est d'ailleurs l'intérêt premier des caches.

On dit que le cache est chaud lorsque les données qui vont être utilisées sont déjà en cache. Lorsque le cache est chaud, la localisation des données dépend seulement de leur taille. Si elles sont suffisamment petites pour tenir dans le L1, elles seront dans le L1.

La performance d'un noyau de calcul en fonction de la taille des données avec un cache chaud permet de visualiser la performance d'un noyau de calcul lorsque les données sont en cache (figure 2.1). On peut alors voir des chutes de performance lorsque les données ne tiennent plus en cache (si le problème est *memory bound*). Ces chutes de performance sont des sorties de caches.

Pour éviter de sortir des caches, il est intéressant d'utiliser des objets plus petits. Par exemple, on peut utiliser des `float`s au lieu de `double`s. L'ensemble des données prendra alors deux fois moins de place, il sera donc possible de traiter des problèmes plus gros tout en gardant les données en cache.

2.1.2.2 Évictions systématiques de cache : *AoSoA*

Le matériel a besoin d'une méthode de correspondance entre adresse en mémoire et position dans le cache. Il en existe principalement trois :
— correspondance directe : une adresse possède une unique position dans le cache,
— associativité complète : une adresse peux être localisée n'importe où dans le cache,
— associativité par ensembles : approche hybride, une adresse peut être localisée dans un unique ensemble de taille fixe.

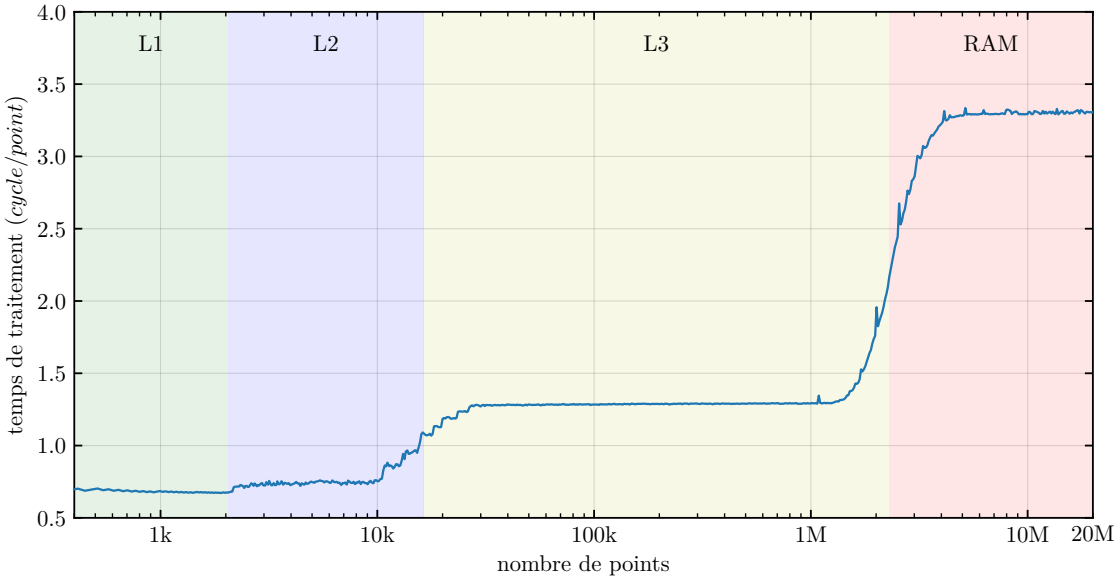


Figure 2.1 – Calcul de barycentre en *SoA* simple précision : sorties de caches (Intel Xeon E5-2683 v3)

Listing 2.8 Tableau de points (*AoSoA*)

```
const int K = /* typiquement la taille d'une ligne de cache */;
struct __Points {
  float x[K], y[K], z[K];
  float w[K];
};
typedef __Points* Points;
```

Listing 2.9 Calcul de barycentre : scalaire (*AoSoA*)

```
Point center_of_mass(Points points, int n) {
  Point center = {0, 0, 0, 0};
  const int N = n / K;
  // n doit être un multiple de K
  for (int I = 0; I < N; I++) {
    for (int i = 0; i < K; i++) {
      center.x += points[I].w[i] * points[I].x[i];
      center.y += points[I].w[i] * points[I].y[i];
      center.z += points[I].w[i] * points[I].z[i];
      center.w += points[I].w[i];
    }
  }
  center.x /= center.w;
  center.y /= center.w;
  center.z /= center.w;
  return center;
}
```

Les caches à associativité complète sont en pratique trop gros à implémenter pour des tailles supérieures à quelques Ko.

Les caches à correspondance directe sont les plus simple à implémenter et plus rapides. Du fait que chaque adresse a une unique position dans le cache, et que le cache est plus petit que la mémoire externe, il faut nécessairement que plusieurs adresses mémoires correspondent à une position dans le cache. Ainsi, le cache ne pourra contenir qu'une seule donnée correspondant à ces adresses conflictuelles : charger une de ces adresses supprimera automatiquement la donnée qui était présente. On parle d'*évictions systématiques* dans ce cas : peut importe l'usage du reste du cache, même "vide", si l'ancienne adresse est très souvent utilisée, la donnée sera évincée.

En pratique, la plupart des caches sont associatifs par ensemble. Un cache N-associatif disposera de plusieurs "ensembles" (*set* en anglais) pouvant contenir chacun jusqu'à N lignes. Il existe une correspondance directe entre adresse et ensemble, mais plusieurs adresses peuvent cohabiter au sein d'un même ensemble. Ce procédé ne résout pas le problème des évictions systématiques, mais l'atténue fortement en autorisant plusieurs adresses dans le même ensemble.

Afin de déterminer l'ensemble associé à une adresse, il est commun d'utiliser les bits de poids faibles de l'adresse comme position de l'ensemble. Dans une telle configuration, si plusieurs adresses partagent les mêmes bits de poids faibles, elles seront associées au même

ensemble. Ceci survient lorsque les adresses sont distantes d'une grande puissance de 2. S'il y a plus de données accédées que l'associativité du cache, on observera des évictions systématiques de cache. Celles-ci se traduisent par des chutes de performances localisées aux puissances de 2.

Si le noyau de calcul utilise plus de pointeurs ou de tableaux (références actives) que l'associativité du cache, le programme risque de subir des évictions systématiques de cache. Ce problème est fortement accentué avec le *multi-threading* où chaque *thread* possède ses propres références actives alors que le cache L3 est partagé entre les *threads*.

De par sa nature, l'agencement mémoire *SoA* requiert plusieurs références actives. Il est ainsi sensible à l'associativité du cache. En revanche, *AoS* n'ayant qu'une seule référence active, il n'a pas cette sensibilité.

Un agencement mémoire hybride est possible, permettant de conserver les avantages de *SoA* concernant la vectorisation tout en n'utilisant qu'une seule référence active comme *AoS* [48]. Un tel agencement est appelé *AoSoA* : il consiste en un tableau de "*SoA*". Pour garder une vectorisation facile et efficace, il faut que le nombre d'éléments en *SoA* soit un multiple de la taille du SIMD. Afin de ne pas tomber dans le même biais que *SoA*, il faut également peu d'éléments en *SoA*. Généralement, une ligne de cache par composante donne de bon résultats.

AoSoA est généralement plus performant, mais est plus difficile à manipuler de par sa nature à plusieurs étages. En effet, une boucle simple écrite en *AoS* ou en *SoA* devra être remplacée par deux boucles imbriquées. De plus, si le nombre d'éléments n'est pas un multiple de la taille des blocs *SoA*, il faut dupliquer le corps de boucle afin de gérer les derniers éléments. Une autre solution est de traiter tous les éléments du dernier bloc inconditionnellement. Les accès mémoires dans ce dernier ne sont pas problématiques car ce bloc est alloué entièrement quoiqu'il arrive. Il faut cependant que les données inutiles soient dans un état valide.

2.1.2.3 Agencements mémoire : récapitulatif

Nous avons vu trois agencements mémoire différents, avec chacun des propriétés différentes (figure 2.2) : *AoS*, *SoA* et *AoSoA*.

AoS est le plus simple d'utilisation (en C). Il n'a qu'une seule référence active, et limite donc les évictions systématiques de cache. De plus, si des éléments du tableau n'ont pas besoin d'être chargés, ceux-ci ne pollueront pas le cache.

En revanche, les composantes non-utilisées des éléments pollueront le cache car à proximité des composantes utilisées. Si notre objet a plusieurs membres qui ne sont pas utilisés par la fonction, comme ceux-ci sont contigus en mémoire (sur la même ligne de

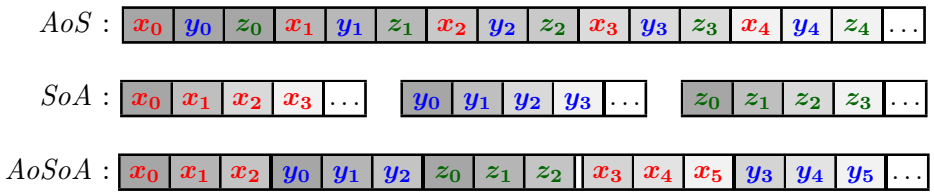


Figure 2.2 – Agencements mémoire : *AoS*, *SoA*, *AoSoA*

cache), ils seront également chargés bien qu'inutiles. Le cache sera donc rempli plus vite car une fraction non-négligeable des éléments qui sont en cache ne sera pas utilisée.

Cet agencement n'est pas adapté à la vectorisation horizontale, il sera donc plus difficile de tirer partie du SIMD.

SoA est un peu plus compliqué à utiliser en C, mais correspond à l'agencement mémoire par défaut en Fortran. Les composantes non-utilisées ne sont jamais mis en cache. Celles-ci sont physiquement éloignées des autres composantes et ne seront donc pas sur les même lignes de cache : ces éléments ne seront jamais chargés en cache. De plus, cet agencement mémoire est particulièrement adapté à la vectorisation horizontale.

Par contre, si tous les objets du tableau ne sont pas accédés, *SoA* va charger la composante de plusieurs objets en cache, même si certains objets ne seront jamais utilisés. En outre, *SoA* requiert plus de références actives (une par composante accédée) et est donc d'avantage sensible aux évictions systématiques de cache. Ce point est d'autant plus important avec l'utilisation du *multithreading* : plusieurs *threads* se partageront le même cache, et donc les ensembles associatifs de ce cache.

AoSoA est un agencement mémoire hybride combinant les caractéristiques du *AoS* et du *SoA*. Il consiste principalement en un tableau de blocs de taille fixe en *SoA*. Tout comme *AoS*, il ne requiert qu'une seule référence active et est donc peu sensible aux évictions systématiques de cache. Tout comme *SoA*, il est particulièrement adapté à la vectorisation horizontale, et permet donc de tirer pleinement partie du SIMD. De la même manière, les composantes non-utilisées ne polluent pas le cache.

Cependant, cet agencement mémoire est plus difficile à manipuler du fait de sa structure à deux niveaux. A l'instar de *SoA*, les éléments non-accédés peuvent polluer le cache.

Ainsi, *AoSoA* a les mêmes propriétés que *SoA*, excepté en ce qui concerne le nombre de références actives qui est réduit à 1 comme en *AoS*. Il est également plus difficile à écrire.

Listing 2.10 Exemple de scalarisation de la multiplication matrice–vecteur

(a) sans scalarisation

```
// Y += L * X
for (i = 0; i < N; i++) {
  for (j = 0; j <= i; j++) {
    Y[i] += L[i][j] * X[j];
  }
}
```

(b) avec scalarisation

```
// Y += L * X
for (i = 0; i < N; i++) {
  float y = Y[i];
  for (j = 0; j <= i; j++) {
    y += L[i][j] * X[j];
  }
  Y[i] = y;
}
```

2.1.3 Scalarisation

La scalarisation consiste à faire persister des données fréquemment utilisées dans des registres. Pour cela, on remplace les expressions avec des accès mémoires par des expressions entre variables, favorisant ainsi leur mise en registre.

La scalarisation peut être effectuée automatiquement par un compilateur optimisant [49], par un optimiseur source à source [50], ou manuellement.

Cette optimisation s'applique particulièrement bien après avoir déroulé complètement une boucle. En effet, une fois qu'une boucle est déroulée complètement, comme le compteur de boucle est devenue une constante, les accès indirects peuvent être calculés directement. On peut dès lors attribuer une variable locale à chaque emplacement mémoire. Les accès intermédiaires ou redondants ne seront donc plus nécessaires. On réduit ainsi fortement les accès mémoire. Cette approche est très similaire au blocage par registres.

Le principal problème de cette technique est l'augmentation drastique de l'utilisation de registres : la pression de registres est importante. Un compilateur risque fortement de générer du *spill code* s'il n'y a pas assez de registres disponibles. L'idée est que le *spill code* utilisera moins souvent et plus efficacement le cache que le code non-scalarisé.

2.1.4 Déroulage de boucle

Le déroulage de boucle est une transformation consistant à répéter le corps de boucle plusieurs fois. Cela peut avoir plusieurs avantages comme diminuer le coût du branchement ou permettre d'augmenter le nombre d'accumulateurs lors des réductions.

Il existe plusieurs types de déroulage de boucles, avec des avantages différents. Nous nous intéresserons à deux déroulages particuliers : le déroulage total (*unwinding* en anglais), et le déroulage avec entrelacement (*unroll&jam* en anglais).

Algorithme 2.1 Exemple de déroulage simple de boucle

(a) Boucle non déroulée

1 **pour** $i = 0 : N$ **faire**
2 $\quad P_i \leftarrow A_i \cdot B_i$
3 $\quad R_i \leftarrow \sqrt{P_i}$

(b) Boucle déroulée d'un facteur 4

1 **pour** $i = 0 : 4 : N$ **faire**
2 $\quad P_{i+0} \leftarrow A_{i+0} \cdot B_{i+0}$
3 $\quad R_{i+0} \leftarrow \sqrt{P_{i+0}}$
4 $\quad P_{i+1} \leftarrow A_{i+1} \cdot B_{i+1}$
5 $\quad R_{i+1} \leftarrow \sqrt{P_{i+1}}$
6 $\quad P_{i+2} \leftarrow A_{i+2} \cdot B_{i+2}$
7 $\quad R_{i+2} \leftarrow \sqrt{P_{i+2}}$
8 $\quad P_{i+3} \leftarrow A_{i+3} \cdot B_{i+3}$
9 $\quad R_{i+3} \leftarrow \sqrt{P_{i+3}}$

Il existe également d'autres transformations de boucles. On notera ainsi les modèles polyhédriques [51] permettant de réorganiser automatiquement l'ordre des itérations sur des boucles imbriquées. De tels modèles sont déjà implémentés dans les compilateurs comme gcc [52, 53].

2.1.4.1 Déroulage simple de boucle

Le déroulage simple consiste à répéter le corps de boucle plusieurs fois (algorithme 2.1). Ainsi, une itération de la nouvelle boucle sera équivalente à plusieurs itérations de la boucle d'origine.

2.1.4.2 Dérouler-Entrelacer : *unroll&jam*

Les processeurs disposent d'un pipeline d'instructions : ils peuvent lancer une instruction avant que la précédente ne soit terminée, si celle-ci ne dépend pas du résultat de la précédente. Il faut donc différencier la latence d'une instruction de son débit. La latence définit le temps (en cycles) qu'il faut attendre avant que le résultat puisse être utilisé par une autre instruction. Le débit (*throughput* en anglais) définit la fréquence à laquelle une instruction peut être exécutée. Par exemple, sur Haswell, une addition flottante a une latence de 3 cycles, et un débit de 1 instr/cycle. Le débit d'une instruction dépend du nombre d'unités fonctionelles pouvant l'exécuter et peut donc être supérieur à 1. Ainsi, une multiplication flottante sur Haswell a un débit de 2 instr/cycle.

De plus, les processeurs peuvent exécuter effectivement plusieurs instructions par cycle tant que ces instructions sont indépendantes et utilisent des unités fonctionnelles différentes (processeurs superscalaires). Pour tirer partie de toute la puissance de tels processeurs, il

Algorithme 2.2 Exemple de déroulage avec entrelacement (*unroll&jam*)

Ce type de déroulage est à comparer avec le déroulage simple (algorithme 2.1)

(a) Boucle non déroulée

1 **pour** $i = 0 : N$ **faire**
2 $\quad P_i \leftarrow A_i \cdot B_i$
3 $\quad R_i \leftarrow \sqrt{P_i}$

(b) Boucle déroulée-entrelacée d'un facteur 4

1 **pour** $i = 0 : 4 : N$ **faire**
2 $\quad P_{i+0} \leftarrow A_{i+0} \cdot B_{i+0}$
3 $\quad P_{i+1} \leftarrow A_{i+1} \cdot B_{i+1}$
4 $\quad P_{i+2} \leftarrow A_{i+2} \cdot B_{i+2}$
5 $\quad P_{i+3} \leftarrow A_{i+3} \cdot B_{i+3}$
6 $\quad R_{i+0} \leftarrow \sqrt{P_{i+0}}$
7 $\quad R_{i+1} \leftarrow \sqrt{P_{i+1}}$
8 $\quad R_{i+2} \leftarrow \sqrt{P_{i+2}}$
9 $\quad R_{i+3} \leftarrow \sqrt{P_{i+3}}$

faut donc que les instructions soient les plus indépendantes possibles, au moins localement. Pour atteindre ce but, les compilateurs réordonnent les instructions d'après le graphe de dépendance. La plupart des processeurs sont également capables de réordonner les instructions dans le même but. On parle d'exécution dans le désordre (*Out-of-Order execution* en anglais).

Toutefois, le réordonnancement d'instructions optimal est NP-difficile [54], et la fenêtre de réordonnancement des processeurs est "petite" : 192 entrées sur Haswell et Ryzen. De par l'utilisation d'heuristique et la visibilité limitée, l'ordre d'exécution est sous-optimal, et le sera d'autant plus que le le chemin critique sera long.

Dérouler-Entrelacer consiste à dérouler partiellement une boucle et à entrelacer les itérations : algorithme 2.2. Si les itérations sont indépendantes, les opérations entrelacées sont également indépendantes les unes des autres. On a donc un graphe de dépendance beaucoup plus large (plus de parallélisme).

Ainsi, on peut augmenter le débit d'instructions grâce à l'indépendance entre les instructions, chaque nouvelle instruction étant prête à être exécutée immédiatement. La performance de la boucle sera donc limitée par le débit des instructions et non leur latence. Cette optimisation est particulièrement intéressante si la latence des instructions est grande alors que leur débit est important : les instructions sont pipelinées. Plus le produit latence×débit des instructions sera haut, plus l'entrelacement sera intéressant. Il en est de même avec la longueur du chemin critique au sein d'une itération.

2.1.4.3 Déroulage total de boucle : *loop unwinding*

Le déroulage total consiste à prendre toutes les itérations d'une boucle, et les mettre à plat, les unes à la suite des autres. Il n'y a ainsi plus aucun branchement, et le compteur de boucle est maintenant une constante différente pour chaque "itération".

Ce déroulage n'est possible que si le nombre total d'itérations de la boucle est petit et connu à la compilation. Comme cela créé autant de copies du corps de boucle que d'itérations au total, il n'est pas conseillé de l'appliquer sur des boucles avec trop d'itérations.

Avec cette optimisation, il est possible de considérer le compteur de boucle comme une constante, et ainsi, de le propager pour réduire la complexité des calculs. Comme la propagation de constantes est une optimisation bien maîtrisée par les compilateurs, de très bons résultats sont obtenus lorsque ce déroulage est appliqué, soit à la main, soit directement par le compilateur.

Cela permet aussi d'éliminer tout branchement lié à la boucle. Ceci peut donner des gains importants sur des boucles imbriquées avec de petits corps de boucles (coût des branchements non-négligeables).

Après déroulage, comme toutes les adresses mémoires sont connues à la compilation (à un décalage près), il est facile d'appliquer la scalarisation. Ainsi, tous les accès mémoire intermédiaires seront supprimés, et seule une lecture unique des données d'entrée et une écriture unique des données de sorties seront nécessaires (listing 2.11). Ces deux transformations se combinent bien avec la fusion d'opérateurs, permettant d'éliminer encore plus d'accès mémoires [55].

Algorithme 2.3 Exemple de déroulage total de boucle

(a) Boucle non déroulée

1 **pour** $i = 0 : 3$ **faire**
2 $\quad P_i \leftarrow A_i \cdot B_i$
3 $\quad R_i \leftarrow \sqrt{P_i}$

(b) Boucle totalement déroulée

1 $P_0 \leftarrow A_0 \cdot B_0$
2 $R_0 \leftarrow \sqrt{P_0}$
3 $P_1 \leftarrow A_1 \cdot B_1$
4 $R_1 \leftarrow \sqrt{P_1}$
5 $P_2 \leftarrow A_2 \cdot B_2$
6 $R_2 \leftarrow \sqrt{P_2}$
7 $P_3 \leftarrow A_3 \cdot B_3$
8 $R_3 \leftarrow \sqrt{P_3}$

Listing 2.11 Déroulage total du produit matrice triangulaire–vecteur avec scalarisation

(a) non déroulé

```
for (i = 0; i < 3; i++) {
  for (j = 0; j <= i; j++) {
    Y[i] += L[i][j] * X[j];
  }
}
```

(b) déroulé et scalarisé

```
// Chargement de X et Y
float x0 = X[0], y0 = Y[0];
float x1 = X[1], y1 = Y[1];
float x2 = X[2], y2 = Y[2];

// Calcul
y0 += L[0][0] * x0;

y1 += L[1][0] * x0;
y1 += L[1][1] * x1;

y2 += L[2][0] * x0;
y2 += L[2][1] * x1;
y2 += L[2][2] * x2;

// Enregistrement de Y
Y[0] = y0;
Y[1] = y1;
Y[2] = y2;
```

2.2 Précision des calculs

2.2.1 Rappels sur le calcul flottant

Il est impossible pour un ordinateur de manipuler des nombres réels. Il faut donc recourir à une approximation des réels. L'approximation la plus utilisée par les ordinateurs est l'arithmétique flottante : [56, 57].

L'ensemble des flottants $\mathbb{F}$ est un sous-ensemble *fini* des réels $\mathbb{R}$. Afin de représenter $x \in \mathbb{R}$, il faut trouver une approximation $\hat{x} \in \mathbb{F}$ telle qu'il n'existe aucun flottant entre x et $\hat{x}$. Cette contrainte ne garantit pas l'unicité de l'approximation. Si $x \notin \mathbb{F}$, il y a 2 valeurs possibles. On peut ainsi définir un mode d'arrondi (une fonction $\mathbb{R} \to \mathbb{F}$) respectant la propriété précédente, et permettant l'unicité de l'approximation. Pour chaque fonction $f : \mathbb{R} \to \mathbb{R}$, il est possible de définir son approximation $\hat{f} : \mathbb{F} \to \mathbb{F}$ telle que $\forall \hat{a} \in \mathbb{F}, \hat{f}(\hat{a}) = \widehat{f(\hat{a})}$.

Une telle approximation ne peut cependant pas garantir $\forall x \in \mathbb{R}, \hat{f}(\hat{x}) = \widehat{f(x)}$: le résultat de la fonction approximante est la meilleure approximation possible uniquement si l'entrée n'est pas déjà une approximation ($x = \hat{x}$). Dans le cas contraire, il n'y a aucune garantie sur la qualité de l'approximation.

La norme IEEE 754 spécifie entièrement une telle arithmétique en définissant le format binaire, les opérations, leur précision... Cette norme définit 3 valeurs spéciales qui ne correspondent à aucun réel : $\{-\infty, +\infty, \text{NaN}\} \subset \mathbb{F}$. Elle définit également 4 modes d'arrondi : vers 0 (troncature), vers $+\infty$, vers $-\infty$ et au plus près. Plusieurs précisions sont supportées : tableau 2.3.

Le terme français "précision" est ambigu et regroupe 2 notions distinctes. La précision des données représente la quantité d'information utilisée pour représenter la valeur (en anglais : *precision*), La précision de calcul désigne la proximité du résultat à la valeur réelle (en anglais : *accuracy*).

Par la suite, le terme "précision" sera utilisé pour désigner la précision des données, et le terme "exactitude" pour désigner la précision de calcul. L'exactitude en bits est donnée par la formule $-\log_2\left(\frac{|\hat{x}-x|}{x}\right)$. On peut également définir l'erreur en bits comme étant la

Table 2.3 – Précision des formats flottants IEEE 754

Précision		demi	simple	double	étendue	quad
type C usuel			`float`	`double`	`long double`	
précision	(bits)	11	24	53	64	113
	(chiffres)	3.3	7.2	15.9	19.3	34.0
exposant maximum		2^{15}	2^{127}	2^{1023}	2^{16383}	2^{16383}
		$10^{4.5}$	$10^{38.2}$	$10^{307.9}$	$10^{4931.7}$	$10^{4931.7}$

différence entre la précision et l'exactitude. Par ailleurs, l'erreur peut aussi s'exprimer en ulp avec la formule suivante : $\frac{|\hat{x}-x|}{x} 2^p$ où p est la précision flottante.

Analyser l'exactitude des résultats en fonction de la précision permet de relâcher les contraintes sur la précision à utiliser afin d'avoir des résultats corrects. Si un calcul en simple précision donne un résultat suffisamment correct (dépendant de l'application), alors il n'est pas nécessaire de recourir à une précision plus haute. On peut donc ainsi accélérer les calculs en utilisant des `float`s plutôt que des `double`s. L'impact sera encore plus important si des flottants plus précis sont utilisés (tel que `long double`), ceux-ci n'étant pas supportés par les instructions SIMD des architectures actuelles.

La question de la reproductibilité des résultats se pose également [58, 59]. En effet, les compilateurs sont capables de faire des transformations qui changent les résultats flottants, mais n'auraient aucun effet en arithmétique réelle. Des options sont disponibles pour activer ou désactiver de telles transformations. Afin d'avoir un code plus rapide, il peut être intéressant d'assouplir les contraintes numériques : augmenter la capacité du compilateur à optimiser au détriment de la reproductibilité.

L'analyse de l'exactitude est conduite en exécutant la fonction à analyser avec des nombres flottants de différentes précisions, et de comparer les résultats avec des flottants plus précis. La bibliothèque MPFR [60] implémente l'arithmétique flottante IEEE 754 en précision arbitraire et peut être utilisée pour conduire cette analyse. La précision arbitraire permet de changer facilement la précision effective des calculs sans changer la fonction. De par le respect de la norme IEEE 754, le comportement de MPFR est identique aux nombres flottants natifs.

Dans ce qui suit, l'exactitude est calculée en comparant les résultats avec une précision donnée et les résultats avec 1024 bits de précision. L'exactitude peut être négative lorsque les calculs sont numériquement instables : le résultat est alors totalement faux et n'est pas du même ordre de grandeur ou du même signe. Elle peut aussi être supérieure à la précision : il n'y a aucun flottant entre le résultat et la valeur réelle. Pour une analyse plus

Table 2.4 – Précision et exactitude : un exemple en binaire

valeur exacte	valeur approchée	précision	exactitude
1.1101_b	1.1_b	2	2.54
1.1101_b	1.11_b	3	4.45
1.1101_b	1.110_b	4	4.45
1.1101_b	1.1101_b	5	∞
1.1101_b	1.11010_b	6	∞
1.1101_b	1.00011_b	6	1.34

poussée de l'exactitude, MPFR n'est pas suffisant et il faut recourir à des outils comme le calcul d'intervalles [61].

En analysant l'exactitude d'un algorithme simple comme la réduction multiplicative (figure 2.3), il apparaît naturellement que l'erreur augmente avec le nombre de multiplications : les erreurs d'arrondis s'accumulent. L'erreur en ulp ne dépends pas de la précision, aux fluctuations aléatoires près : en effet, l'erreur se propage depuis les bits de poids faibles. Ainsi, si un résultat plus précis de n bits est requis et que les entrées sont suffisamment précises, il suffit alors d'augmenter la précision des calculs de n bits également.

La principale source d'instabilité numérique n'est cependant pas les erreurs d'arrondis dues aux multiplications, mais la perte de précision (*significance loss*) due aux additions et soustractions.

La perte de précision apparaît lorsque une addition ou une soustraction ne peut être calculé sans perdre de bits corrects. Il en existe deux variantes :

Absorption : Lorsqu'un petit nombre est additionné à un grand nombre (figure 2.4a). Dans ce cas, les bits de poids faible du petit nombre seront ignorés. Avec des nombres à 5 chiffres, cela donne : $1234.5 + 1.2345 \rightarrow 1235.7$ au lieu de 1235.7345. Répétée, l'absorption peut conduire à de grandes erreurs.

Cancellation : Lorsque deux nombres proches sont soustrait l'un de l'autre (figure 2.4b). Le résultat est calculé avec moins de bits de précision. Avec des nombres à 5 chiffres, cela donne : $1235.7 - 1234.5 \rightarrow 1.2$. Le résultat possède une précision de 2 chiffres.

Si l'un des nombres est déjà une approximation, l'effet sera bien plus visible. En effet, si 1234.6 est l'approximation de $1234.5 + 1.2345$, le calcul complet donnera $((1234.5 + 1.2345) - 1234.5) \rightarrow 1.2$ au lieu du résultat correct 1.2345 qui tient sur 5 chiffres. Quand il ne reste qu'une poignée de bits, on parle de *catastrophic cancellation*.

L'algorithme utilisé peut avoir une influence sur l'exactitude du résultat. On peut ainsi mentionner la sommation de Kahan [62] qui utilise l'arithmétique compensée afin de réduire l'erreur.

2.2.2 Racine carrée inverse

Le calcul de la racine carrée inverse ($f(x) = 1/\sqrt{x}$) est une partie importante et incontournable de la factorisation de Cholesky. Sur la plupart des architectures, la division et la racine carrée sont des opérations lentes, et sont traitées par la même unité fonctionnelle. Par exemple, sur Skylake-X (tableau 2.5), il n'est possible de lancer une division ou une racine carrée qu'une fois tous les 10 à 12 cycles (en AVX512) : ces instructions ne sont que faiblement pipelinées. Il est donc possible d'effectuer une racine carrée inverse tous les

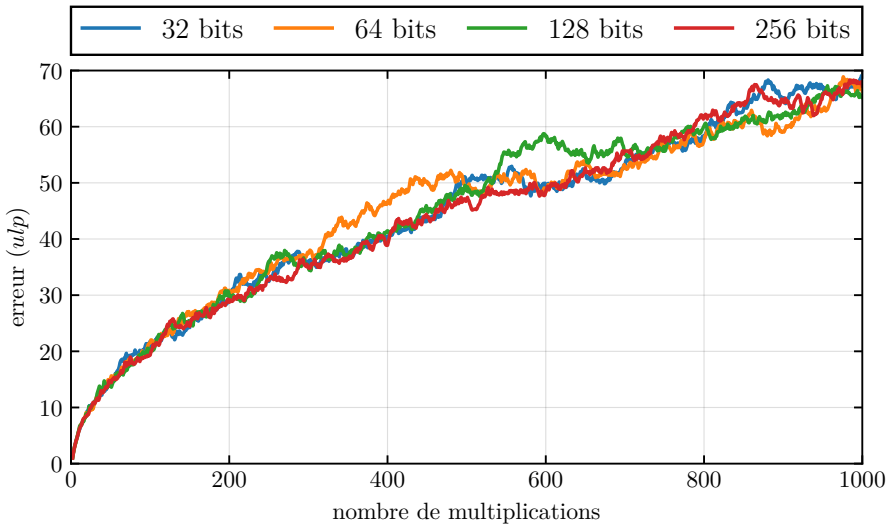


Figure 2.3 – Erreur de la réduction multiplicative en ulp

	1 2 3 4.5
+	1.2 3 4 5
exact	1 2 3 5.7 3 4 5
arrondi	1 2 3 5.7

(a) Exemple d'absorption

	1 2 3 5.7
-	1 2 3 4.5
exact	1.2
arrondi	1.2 0 0 0

(b) Exemple de cancellation

Figure 2.4 – Exemple de perte de précision en arithmétique flottante à 5 chiffres (*significance loss*)

Table 2.5 – Latence et débit inverse en cycle des instructions flottantes usuelles (Skylake-X) [63]

latence/débit^{-1}	128 bits (SSE)	256 bits (AVX)	512 bits (AVX512)
···`_add_ps`	4/0.5	4/0.5	4/ 0.5
···`_mul_ps`	4/0.5	4/0.5	4/ 0.5
···`_fmadd_ps`	4/0.5	4/0.5	4/ 0.5
···`_rcp_ps`	4/1	4/1	7/ 2
···`_div_ps`	11/3	11/5	18/10
···`_div_pd`	13/4	13/8	24/16
···`_rsqrt_ps`	4/1	4/1	6/ 2
···`_sqrt_ps`	12/3	12/6	20/12
···`_sqrt_pd`	15/4	15/9	28/18

22 cycles. À noter que l'on parle ici de débit et non de latence qui approche les 38 cycles. À titre de comparaison, en 22 cycles, le même processeur Skylake-X est capable d'effectuer 44 FMA, soit 88 opérations arithmétiques.

Il existe cependant des méthodes pour approximer rapidement la racine carrée inverse [64], et ainsi accélérer le code. Il sera alors possible de raffiner l'approximation avec des algorithmes itératifs tels la méthode de Newton-Raphson.

Tout ceci peut également s'appliquer au calcul de l'inverse ($f(x) = 1/x$), avec des gains cependant plus limités : le calcul de l'inverse est plus rapide que celui de la racine carré inverse.

2.2.2.1 Première approximation

Instruction dédiée. La plupart des architectures proposent directement une instruction afin d'approximer la racine carrée inverse en `float`. Celle-ci est généralement aussi rapide qu'une multiplication flottante et entièrement pipelinée. Excepté en Neon, cette instruction donne au moins 12 bits corrects (tableau 2.6). Les Xeon Phi (Knights Corner et Knights Landing) sont capables de calculer la racine carrée inverse correcte jusqu'au dernier bit à peine plus lentement qu'une multiplication (7 à 8 cycles de latence contre 6).

Matériellement, une telle instruction peut utiliser un *bit-shift* (décalage) pour le calcul de l'exposant, et une *lookup-table* (table de correspondance) pour la mantisse.

En AVX512, Neon 64 bits et VSX, ces instructions sont également disponibles en `double`. Pour les autres architectures, il est possible de convertir l'entrée en `float`, d'appeler l'instruction en `float`, puis de reconvertir en `double` le résultat (algorithme 2.4). En SIMD, cette conversion nécessite de gérer des registres de tailles différentes. De ce fait, il est possible d'appliquer une seule instruction en `float` agissant sur deux registres en `double` (avant conversion).

Cette technique ne peut cependant pas s'appliquer si l'entrée n'est pas dans la plage de valeurs couverte par le format simple précision : $[2^{-126}, 2^{127}]$ ($[\simeq 10^{-38}, \simeq 10^{38}]$). Si la plage

Algorithme 2.4 Approximation de RSQRT en double précision via la simple précision

entrée : $x_{0,F64}$, $x_{1,F64}$
sortie : $\hat{r}_{0,F64}$, $\hat{r}_{1,F64}$ // approximation $1/\sqrt{x}$
1 $\text{low}(x_{F32}) \leftarrow \text{convert_f64_to_f32}(x_{0,F64})$
2 $\text{high}(x_{F32}) \leftarrow \text{convert_f64_to_f32}(x_{1,F64})$
3 $\hat{r}_{F32} \leftarrow \text{rsqrte}(x_{F32})$ // approximation à 12 bits
4 $\hat{r}_{0,F64} \leftarrow \text{convert_f32_to_f64}(\text{low}(\hat{r}_{F32}))$
5 $\hat{r}_{1,F64} \leftarrow \text{convert_f32_to_f64}(\text{high}(\hat{r}_{F32}))$

Table 2.6 – Instruction pour l'approximation de la racine carrée inverse en simple précision

ISA	Nom de l'*intrinsic*	Erreur relative maximale	Machines
Altivec	`vec_rsqrte`	$< 2^{-12}$	$\geqslant$ PowerPC G4 $\geqslant$ Power 6
VSX	`vec_rsqrte`	$< 2^{-14}$	$\geqslant$ Power 7
Neon	`vrsqrteq_f32`	$< 2^{-8}$	$\geqslant$ Cortex A8
SSE	`_mm_rsqrt_ps`	$< 1.5 \times 2^{-12}$	$\geqslant$ Pentium 4
AVX	`_mm256_rsqrt_ps`	$< 1.5 \times 2^{-12}$	$\geqslant$ Sandy Bridge
KNCNI	`_mm512_rsqrt23_ps`	< 0.5 ulp	KNC
AVX512F	`_mm512_rsqrt14_ps`	$< 2^{-14}$	Skylake Xeon
AVX512ER	`_mm512_rsqrt28_ps`	< 0.5 ulp	KNL

des valeurs du format simple précision n'est pas suffisante, il faut alors recourir à une autre méthode. Toutefois, cette plage peut être suffisante dans de nombreux domaines comme le traitement du signal, le traitement d'images, la vision par ordinateur ou la Physique des Hautes Énergies.

Manipulation de bits. La racine carrée inverse peut également être approximée en utilisant la représentation binaire d'un nombre en virgule flottante spécifiée par la norme IEEE 754 (algorithme 2.5). Cette technique a été initialement attribuée à John Carmack pour son utilisation dans le code de Quake 3 (en simple précision). Le fondement mathématique de celle-ci est expliqué par Chris Lomont [65].

Il y a 3 points cruciaux :
- le décalage à droite permet de diviser par 2 l'exposant du nombre (racine carrée),
- la soustraction permet de prendre l'opposé de l'exposant (inverse),
- les bits de poids faible de la constante permettent de minimiser l'erreur sur la mantisse.

Les deux premiers points imposent les bits correspondants au signe et à l'exposant :

Algorithme 2.5 Approximation de RSQRT par manipulation de bits

entrée : x_{F64}
sortie : $\hat{r}_{F64}$ // approximation de $1/\sqrt{x}$
1 $x_{I64} \leftarrow$ cast_f64_to_i64(x_{F64})
2 $\hat{r}_{I64} \leftarrow$ `0x5fe6ec85e7de30da` $-$ ($x_{I64} >> 1$)
3 $\hat{r}_{F64} \leftarrow$ cast_i64_to_f64($\hat{r}_{I64}$)

`0x5fe0000000000000`. L'optimisation des bits de la mantisse peut se faire par force brute (dans le cas de la simple précision), ou par une analyse mathématique et statistique. L'analyse mathématique de Chris Lomont donne la constante suivante : `0x5fe6ec85e7de30da`.

Cette approche est très rapide, en particulier si les opérations entières et flottantes peuvent être exécutées en parallèle. Elle est cependant approximative avec une erreur relative maximale de l'ordre de $0.0342128 \simeq \frac{1}{29}$.

2.2.2.2 Raffinement de l'approximation

Selon les applications, les approches précédentes peuvent être trop approximatives, particulièrement la manipulation de bits. Il est possible de d'augmenter l'exactitude [66] de l'approximation par des algorithmes itératifs comme la méthode de Newton-Raphson ou sa généralisation d'ordre supérieur : la méthode de Householder.

Le nombre d'itérations à appliquer est à adapter à l'exactitude voulue afin d'être plus rapide [67].

Newton-Raphson. La méthode de Newton-Raphson est un algorithme itératif permettant de trouver les zéros d'une fonction $f(x)$. Étant donnée une approximation x_n d'un zéro de la fonction f, il est possible de calculer une meilleure approximation x_{n+1} grâce à la formule suivante :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.1}$$

Cette méthode est utilisable pour améliorer une approximation de la racine carrée inverse. En effet, pour calculer la racine carrée inverse d'un nombre a, on peut chercher les zéros de la fonction $f(x) = \frac{1}{x^2} - a$.

En appliquant 2.1 à cette fonction, on obtient la formule suivante :

$$x_{n+1} = \frac{1}{2}\, x_n \left(3 - {x_n}^2\, a\right) \tag{2.2}$$

D'après l'équation 2.2, chaque itération nécessite 4 multiplications (algorithme 2.7). La multiplication par $\frac{1}{2}$ peut être injectée à l'intérieur des parenthèses, et le produit $\frac{1}{2} \cdot a$ calculé une seule fois avant toute itération. Ainsi, il est possible d'économiser une multiplication par itération, avec une multiplication supplémentaire à l'initialisation. Il est possible d'utiliser un FMA si l'architecture le supporte.

La méthode de Newton-Raphson a une convergence quadratique : le nombre de bits corrects est doublé à chaque itération (ε devient ε^2). Lorsque l'on applique cette méthode à l'approximation de `_mm_rsqrt_ps`, une seule itération permet d'obtenir un résultat précis avec une erreur moyenne < 0.5 ulp et une erreur maximale < 2.7 ulp (moins de 2 bits).

Algorithme 2.6 Erreur relative de `rsqrt`(x) en ulp

entrée : x_{F32}
sortie : ϵ_{32}

1 // calcul d'une approximation à 12 bits suivie d'une itération de Newton-Raphson (F32)
2 $\hat{r}_{F32} \leftarrow$ `rsqrt`(x)
3 $x_{F64} \leftarrow$ convert_f32_to_f64(x)
4 $\hat{r}_{F64} \leftarrow 1/\sqrt{x_{F64}}$ // calcul précis (F64)
5 $\hat{r}_{I64} \leftarrow$ cast_f64_to_i64$(\hat{r}_{F64})$
6 $\hat{r}_{F32\rightarrow F64} \leftarrow$ convert_f32_to_f64$(\hat{r}_{F32})$
7 $\hat{r}_{F32\rightarrow I64} \leftarrow$ castf64_to_i64$(\hat{r}_{F32\rightarrow F64})$
8 $\epsilon_{64} \leftarrow |\hat{r}_{I64} - \hat{r}_{F32\rightarrow I64}|$ // distance (F64)
9 $\epsilon_{32} \leftarrow \epsilon_{64}/2^{53-24}$ // ulp (F32)

Table 2.7 – Nombre d'opérations pour le raffinement de RSQRT avec la méthode de Newton-Raphson en fonction de la précision source et cible

précision source	précision cible	#iter	#FMA	#mul	#add	**#op**
1/29	2^{-24} (F32)	3	3	7	-	**10**
2^{-8}	2^{-24} (F32)	2	2	5	-	**7**
2^{-12}	2^{-24} (F32)	1	1	3	-	**4**
2^{-14}	2^{-24} (F32)	1	1	3	-	**4**
1/29	2^{-53} (F64)	4	4	9	-	**13**
2^{-8}	2^{-53} (F64)	3	3	7	-	**10**
2^{-12}	2^{-53} (F64)	3	3	7	-	**10**
2^{-14}	2^{-53} (F64)	2	2	5	-	**7**
2^{-23}	2^{-53} (F64)	2	2	5	-	**7**
2^{-28}	2^{-53} (F64)	1	1	3	-	**4**
initialisation			-	1	-	**1**
iteration			1	2	-	**3**

Algorithme 2.7 Newton-Raphson pour le calcul de $1/\sqrt{x}$

entrée : x
entrée : $\hat{r}$ // approximation de $1/\sqrt{x}$
sortie : r // meilleure approximation
1 $\alpha \leftarrow \hat{r} \cdot \hat{r} \cdot x$
2 $r \leftarrow 0.5 \cdot \hat{r} \cdot (3 - \alpha)$

Algorithme 2.8 Newton-Raphson pour le calcul de $1/\sqrt{x}$ écrit en Neon

entrée : x
entrée : $\hat{r}$ // approximation de $1/\sqrt{x}$
sortie : r // meilleure approximation
1 $\alpha \leftarrow$ `vrsqrtsq_f32`$(\hat{r} \cdot x, \hat{r})$
2 $r \leftarrow \hat{r} \cdot \alpha$

L'erreur maximale et relative sont calculées exhaustivement avec l'algorithme 2.6 appliqué à tous les nombres flottants simple précision normaux. (voir tableau 2.7 pour la double précision)

Le jeu d'instruction Neon possède une instruction pour accélérer Newton-Raphson appliqué à $f(x) = \frac{1}{x^2} - a$ (algorithme 2.8). L'instruction `vrsqrtsq_f32` est aussi rapide qu'une multiplication et permet d'économiser 2 multiplications et 1 soustraction (ou 1 FMA et 1 multiplication). L'intérêt n'est pas seulement d'économiser quelques instructions, mais aussi ne plus avoir besoin des deux constantes (0.5 et 3).

Householder. La méthode de Householder est une généralisation d'ordre supérieur de la méthode de Newton-Raphson. La vitesse de convergence dépend de l'ordre choisi pour appliquer la méthode :

— ordre 1 : convergence quadratique (méthode de Newton-Raphson)
— ordre 2 : convergence cubique (méthode de Halley)
— ordre 3 : convergence quartique

Sebah [68] explique comment appliquer la méthode de Householder d'ordre quelconque pour une fonction f :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \left(1 + \frac{f(x_n) f''(x_n)}{2!\ f'(x_n)^2} + \frac{f(x_n)^2 \left(3\ f''(x_n)^2 - f'(x_n) f^{(3)}(x_n) \right)}{3!\ f'(x_n)^4} + \cdots \right) \tag{2.3}$$

Algorithme 2.9 Householder appliqué au calcul de $1/\sqrt{x}$

entrée : x
entrée : $\hat{r}$ // approximation de $1/\sqrt{x}$
sortie : r // meilleure approximation
1 $\alpha \leftarrow \hat{r} \cdot \hat{r} \cdot x$
2 $r \leftarrow \hat{r} \cdot (\frac{35}{16} - \alpha \cdot (\frac{35}{16} - \alpha \cdot (\frac{21}{16} - \frac{5}{16} \cdot \alpha)))$
3 // Ces fractions sont représentables exactement en flottant et n'introduisent donc aucune erreur

Table 2.8 – Raffinement de RSQRT avec la méthode de Householder

précision source	précision cible	ordre	#iter	#FMA	#mul	**#op**
1/29	2^{-24} (F32)	4	1	4	3	**7**
2^{-8}	2^{-24} (F32)	2	1	2	3	**5**
2^{-12}	2^{-24} (F32)	1	1	1	3	**4**
2^{-14}	2^{-24} (F32)	1	1	1	3	**4**
1/29	2^{-53} (F64)	3	2	6	6	**12**
2^{-8}	2^{-53} (F64)	5	1	5	3	**8**
2^{-12}	2^{-53} (F64)	4	1	4	3	**7**
2^{-14}	2^{-53} (F64)	3	1	3	3	**6**
2^{-23}	2^{-53} (F64)	2	1	2	3	**5**
2^{-28}	2^{-53} (F64)	1	1	1	3	**4**

Toutes les additions sont calculées avec des FMA

Comme pour Newton-Raphson, on recherche les zéros de la fonction $f(x) = \frac{1}{x^2} - a$. En s'arrêtant au troisième ordre, on obtient la formule suivante :

$$x_{n+1} = x_n \left(\frac{35}{16} - \frac{35}{16}\, {x_n}^2\, a + \frac{21}{16}\, {x_n}^4\, a^2 - \frac{5}{16}\, {x_n}^6\, a^3 \right) \tag{2.4}$$

On remarque dans 2.4 que dans les parenthèses se trouve un polynôme en ${x_n}^2\, a$. Ce qui nous donne la formule suivante :

$$\begin{aligned} \alpha_n &= {x_n}^2\, a \\ x_{n+1} &= x_n \left(\frac{35}{16} - \frac{35}{16}\, \alpha_n + \frac{21}{16}\, {\alpha_n}^2 - \frac{5}{16}\, {\alpha_n}^3 \right) \end{aligned} \tag{2.5}$$

Le schéma de Horner permet de calculer la valeur d'un polynôme en utilisant un nombre minimal de multiplications [69].

$$\begin{aligned} \alpha_n &= {x_n}^2\, a \\ x_{n+1} &= x_n \left(\frac{35}{16} + \alpha_n \left(\frac{-35}{16} + \alpha_n \left(\frac{21}{16} + \alpha_n\, \frac{-5}{16} \right) \right) \right) \end{aligned} \tag{2.6}$$

En l'appliquant à un polynôme de degré n, il faut n multiplications et n additions. De plus, ces multiplications et additions peuvent être fusionnées en n FMA. Si l'architecture le supporte, on peut donc évaluer un polynôme uniquement avec des FMA qui sont aussi rapides que des multiplications.

On peut donc implémenter la racine carré inverse efficacement avec l'ordre 3 de la méthode de Householder qui requiert seulement 3 FMA et 3 multiplications. C'est une multiplication de moins qu'avec la méthode de Newton-Raphson.

Il est possible de mener ces calculs pour les autres ordres. On peut alors voir, selon l'exactitude de départ et l'exactitude recherchée, quel ordre permet de calculer la racine carrée inverse avec le moins d'opérations. Le tableau 2.8 récapitule les complexités de calculs jusqu'à l'ordre 5.

2.3 Génération de code

La génération de code est une méthode d'abstraction du code. Cette abstraction du code ne s'acompagne pas de coût en terme de vitesse de traitement, et est donc bien adaptée à l'accélération de codes. La génération de code peut ainsi être utilisée pour avoir un code GPU compatible à la fois avec OpenCL et Cuda [70].

Le code est généré à partir de patrons (communément appelés *template* dans la littérature anglophone). Ces patrons sont traités par le moteur de patrons (*template engine*) Jinja2 [71], écrit en Python.

Jinja2 utilise un langage de balisage à l'instar de PHP afin de d'entremêler le flux de contrôle du générateur au sein du patron : le C est entremêlé de Python contrôlant la génération (tout comme le HTML est entremêlé de PHP). Jinja2 n'a aucune connaissance de la syntaxe du C : il effectue juste des opérations sur le texte brut.

Une structure de type `{{expr}}` va interpréter *expr* comme une expression Python, l'exécuter et copier le résultat dans la sortie du générateur. Les structures de type `{% ... %}` sont des primitives Jinja2 et sont utilisées pour écrire des conditions, des boucles ou des macros.

Avec un tel moteur de patrons, il est possible d'implémenter certaines optimisations précédemment présentées : principalement le déroulage, la scalarisation et le SIMD. Ces transformations sont également implémentables directement avec les *templates* C++ [72], mais ceux-ci sont plus difficiles à utiliser et ne permettent pas de voir le code après ces transformations.

2.3.1 Déroulage total

Dérouler complètement une boucle est une tâche facile en Jinja2 (listing 2.13). Il suffit d'utiliser les boucles natives de Jinja2 qui permettent de répéter un bloc de texte un certain nombre de fois en changeant la valeur du compteur de boucle (qui peut être utilisé pour affecter la génération du bloc).

Listing 2.12 Boucle `for` en C

```
for (int i = 0; i < 4; i++) {
  s    = A[i] * B[i];
  C[i] = sqrt(s);
}
```

Listing 2.13 Déroulage total avec Jinja2

(a) Code source

```
{% for i in range(4) %}
s{{i}}   = A[{{i}}] * B[{{i}}];
C[{{i}}] = sqrt(s{{i}});
{% endfor %}
```

(b) Code généré

```
s0   = A[0] * B[0];
C[0] = sqrt(s0);
s1   = A[1] * B[1];
C[1] = sqrt(s1);
s2   = A[2] * B[2];
C[2] = sqrt(s2);
s3   = A[3] * B[3];
C[3] = sqrt(s3);
```

Comme Jinja2 agit sur le texte brut, il est possible (et facile) de créer de nouvelles variables. Dans l'exemple précédent, la variable locale `s` utilisé en C possède maintenant une version par itération déroulée : `s0`, `s1`, `s2`, `s3` qui sont effectivement des variables différentes. La boucle est ainsi complètement déroulée et scalarisée : il n'y a aucun accès mémoire effectué pour les résultats temporaires.

Toutefois, bien que celle-ci soit simplifiée par l'utilisation de Jinja2, la scalarisation n'est pas automatique et requiert tout de même des changements au code.

2.3.2 Dérouler-entrelacer et scalarisation

Il est également possible de "dérouler-entrelacer" en Jinja2 : listing 2.14. On voit facilement que la sortie est très proche de listing 2.13 à ce détail près : les lignes ne sont pas dans le même ordre. Dans le cas de l'entrelacement, toutes les lignes correspondant à une même ligne dans le patron sont groupées ensembles. Alors que dans le déroulage total (sans entrelacement), c'est le bloc entier qui est répété. Toutefois, ceci n'est pas natif : il faut recourir à un filtre. Un filtre est une fonction Python qui est appliquée à un bloc de texte, après que celui-ci ait été généré par Jinja2.

Le filtre `unrollNjam` va prendre une liste de symboles, et dupliquer toutes les lignes du bloc contenant le caractère `@` autant de fois qu'il y a de symboles dans la liste. Le `@` sera alors remplacé par les symboles de la liste : les `@` de la première copie seront remplacés par

Listing 2.14 Entrelacement avec Jinja2

(a) Code source

```
{% filter unrollNjam(range(4)) %}
s@   = A[@] * B[@];
C[@] = sqrt(s@);
{% endfor %}
```

(b) Code généré

```
s0   = A[0] * B[0];
s1   = A[1] * B[1];
s2   = A[2] * B[2];
s3   = A[3] * B[3];
C[0] = sqrt(s0);
C[1] = sqrt(s1);
C[2] = sqrt(s2);
C[3] = sqrt(s3);
```

le premier symbole, ceux de la deuxième par le deuxième symbole et ainsi de suite. Les lignes ne contenant aucun `@` ne seront pas touchées.

Le caractère `@` est le seul caractère ASCII qui n'a aucune signification en C ou C++ et dont l'utilisation est invalide. Bien que les standards donnent le même statut au caractère `$`, celui-ci est très souvent accepté par les compilateurs dans les noms de variables. Le caractère `@` est donc le candidat parfait pour cette tâche.

Pour un petit exemple avec toutes les fonctionnalités de Jinja2 vues précédemment, le listing 2.15 présente le code d'un produit matrice triangulaire–vecteur de dimension 3.

Listing 2.15 Produit Matrice triangulaire-vecteur en *SoA* avec Jinja2

(a) Code source

```
{% filter unrollNjam(range(2)) %}
{%   for i in range(3) %}
x{{i}}_@ = X[{{i}}][@];
{%     for j in range(i+1) %}
{%       if j == 0 %}
s@  = L[{{i}}][{{j}}][@] * x{{j}}_@;
{%       else %}
s@ += L[{{i}}][{{j}}][@] * x{{j}}_@;
{%       endif %}
{%     endfor %}
Y[{{i}}][@] = s@;
{%   endfor
{% endfor %}
```

(b) Code généré

```
x0_0 = X[0][0];
x0_1 = X[0][1];
s0  = L[0][0][0] * x0_0;
s1  = L[0][0][1] * x0_1;
Y[0][0] = s0;
Y[0][1] = s1;
x1_0 = X[1][0];
x1_1 = X[1][1];
s0  = L[1][0][0] * x0_0;
s1  = L[1][0][1] * x0_1;
s0 += L[1][1][0] * x1_0;
s1 += L[1][1][1] * x1_1;
Y[1][0] = s0;
Y[1][1] = s1;
x2_0 = X[2][0];
x2_1 = X[2][1];
s0  = L[2][0][0] * x0_0;
s1  = L[2][0][1] * x0_1;
s0 += L[2][1][0] * x1_0;
s1 += L[2][1][1] * x1_1;
s0 += L[2][2][0] * x2_0;
s1 += L[2][2][1] * x2_1;
Y[2][0] = s0;
Y[2][1] = s1;
```

2.3.3 SIMD

La génération de code SIMD portable est gérée par un préprocesseur écrit en Python. Celui-ci s'interface avec Jinja2 au travers d'objets Python spéciaux, nommés "macros", utilisables dans les patrons.

Quand une telle macro est utilisée au sein du patron (listing 2.16), le moteur de patron la remplace par un identifiant unique, sans toucher au contenu des parenthèses (listing 2.16b). Puis, le préprocesseur fait une analyse lexicale du texte brut généré : il le découpe en lexèmes. Lorsque le préprocesseur reconnaît le lexème d'une macro (un des identifiants unique générés précédemment), il va lire tous les lexèmes à l'intérieur des parenthèses, puis appliquer effectivement la macro à cette liste de lexèmes (listing 2.16c). La suite de lexèmes est ensuite réassemblée en texte brut pour écrire le fichier final.

Le code généré dépend du jeu d'instructions ciblé (listing 2.16c vs listing 2.16d). Avec ces deux exemples, on peut voir que le code généré est complètement différent : les noms, le nombre d'arguments, ainsi que l'ordre des arguments des fonctions sont différents. C'est pourquoi une approche lexicale est requise : une simple substitution textuelle n'aurait pas permis de changer l'ordre des arguments par exemple.

La définition d'une macro se fait très simplement en Python (listing 2.17). Le décorateur `@macro` est cependant assez complexe et fortement intriqué avec le préprocesseur lui-même.

Une macro peut être définie à partir d'une chaîne de caractère : la suite de lexèmes est traduite en texte brut, puis la chaîne de caractères renvoyée par la fonction est retraduite en liste de lexèmes par une nouvelle analyse lexicale. Une macro peut aussi être définie en fonction d'autres macros (comme la macro `accmul` de l'exemple). Dans ce cas, la macro peut être appliquée sans repasser par le texte brut.

Les différents jeux d'instructions sont gérés par des classes distinctes qui implémentent les macros différemment.

Cette approche est suffisamment souple pour générer non seulement du code SIMD utilisant les *intrinsics*, mais aussi pour utiliser des bibliothèques SIMD.

Listing 2.16 Exemple d'utilisation de macros

(a) Code source

```
{{vec}} h, a, b;
h = {{vec(0.5)}};
a = {{vec.load}}(&A[i]);
b = {{vec.mul}}(a, h);
{{vec.store}}(&B[i], b);
```

(b) Code généré avant préprocesseur

```
vector float h, a, b;
h = $altivec$vecf32x4$splat$(0.5f);
a = $altivec$vecf32x4$load$(&A[i]);
b = $altivec$vecf32x4$mul$(a, h);
$altivec$vecf32x4$store$(&B[i], b);
```

(c) Code généré (Altivec)

```
vector float h, a, b;
h = ((vector float){0.5f,0.5f,0.5f,0.5f});
a = vec_ld(0, &A[i]);
b = vec_madd(a, h, (vector float){});
vec_st(b, 0, &B[i]);
```

(d) Code généré (SSE)

```
__m128 h, a, b;
h = _mm_set1_ps(0.5f);
a = _mm_load_ps(&A[i]);
b = _mm_mul_ps(a, h);
_mm_store_ps(&B[i], b);
```

Listing 2.17 Définition de macros pour le SIMD

```
class SSE(x86Vec):
  @macro
  def add(self, a, b):
    return "_mm_add_ps({}, {})".format(a, b)

  @macro
  def mul(self, a, b):
    return "_mm_mul_ps({}, {})".format(a, b)

  @macro
  def accmul(self, a, b, c):
    return self.add(a, self.mul(b, c))
```

2.4 Synthèse

Dans cette partie, nous avons vu et expliqué différentes transformations après un bref historique des architectures SIMD. Ainsi, nous avons fait la distinction entre vectorisation verticale (au sein d'un même élément) et la vectorisation horizontale (sur plusieurs éléments indépendants). La vectorisation horizontale est plus efficace et passe mieux à l'échelle (avec du SIMD plus grand) lorsque les éléments sont petits ou que les composantes nécessitent des traitements différents.

Nous avons également vu plusieurs agencements mémoires avec des propriétés relatives au SIMD et au cache différentes. Ainsi, l'agencement mémoire *AoS* n'est pas adapté à la vectorisation horizontale, mais n'utilise qu'une référence active et est donc peu sensible aux évictions systématiques de cache. L'agencement *SoA* est bien adapté à la vectorisation horizontale, mais nécessite plusieurs références actives ce qui le rend sensible aux évictions systématiques de cache. L'agencement *AoSoA* est hybride entre le *AoS* et *SoA*. Comme *SoA*, il est bien adapté à la vectorisation horizontale, et comme *AoS*, il est peu sensible aux évictions systématiques de cache.

La scalarisation permet d'augmenter l'intensité arithmétique en persistant les variables en registres et s'affranchissant ainsi des accès mémoires. La scalarisation est particulièrement efficace après avoir déroulé totalement les petites boucles où le nombre d'itérations est fixe. Dérouler–Entrelacer permet de s'affranchir des latences des instructions en augmentant le parallèlisme local. Ces approches augmentent cependant la pression de registre ce qui peut conduire à la génération de *spill code* par le compilateur.

Nous avons ensuite rappelé les fondements de l'arithmétique flottante IEEE 754. Nous avons présenté la distinction entre la précision qui correspond à la quantité d'information utilisée pour encoder un nombre, et l'exactitude qui correspond à la proximité d'un résultat à la valeur réelle.

Nous avons présenté les différents phénomènes à l'origine de la perte de précision : l'accumulation d'erreurs d'arrondi, l'absorption (lorsqu'un nombre très petit est additionné à un nombre très grand) et la cancellation (lorsque deux nombres très proches sont soustraits).

Nous avons ensuite présenté le calcul rapide de la racine carrée inverse. Ce calcul rapide consiste en une première approximation (soit par une instruction spécialisée, soit par la manipulation des bits directement), suivie par un raffinement à l'aide de la méthode de Newton-Raphson, ou de sa généralisation d'ordre supérieur : la méthode de Householder.

Enfin, nous avons présenté un générateur de code basé sur le moteur de patrons Jinja2 en Python. Ce générateur nous permet d'écrire facilement la plupart des transformations vues précédemments, mais aussi d'écrire du SIMD avec des *intrinsics*, et ce, de manière portable.

Chapitre 3

Études préliminaires : calcul de paraboles

Dans cette partie, nous étudierons des algorithmes simples de la reconstruction LHCb. Le but est de se familiariser avec les transformations précédentes et de mettre en place une méthode pour mesurer de manière fiable la performance des algorithmes, et leur précision de calculs.

La méthode ainsi définie sera utilisée dans les chapitres suivants afin de simplifier l'analyse de résultats futurs.

3.1 Présentation des algorithmes

Le premier algorithme étudié permet de calculer les paramètres d'une parabole passant par trois points : algorithme 3.1. Celui-ci est utilisé lors de la reconstruction des trajectoires des particules afin de rendre compte de la courbure induite par le champs magnétique.

En pratique, le code résout une équation cubique où le dernier paramètre est fixé. Dépendant de la configuration, ce paramètre peut être égal à zéro : l'interpolation est alors quadratique.

Le deuxième algorithme est proche, mais considère tous les points d'une trajectoire : algorithme 3.2. Le problème devient donc un problème d'optimisation où le but est de trouver la parabole la plus proche de la trajectoire selon l'erreur quadratique. Cet algorithme est itératif et requiert une première approximation des paramètres qui ont été calculés auparavant lors de la reconstruction effective de la trace.

Le but principal ici est de calculer la vraisemblance de la trajectoire par un test du χ^2, tout en raffinant les paramètres de la trajectoire.

Algorithme 3.1 `solveParabola`

global : z_{ref}
entrée : $x_1,\ x_2,\ x_3$ // Coordonnée X des 3 points
entrée : $z_1,\ z_2,\ z_3$ // Coordonnée Z des 3 points
entrée : d_r // Paramètre cubique fixé
sortie : $a,\ b,\ c$ // Paramètres de la parabole

1 $r_1 \leftarrow 1 + d_r\,(z_1 - z_{ref})$
2 $r_2 \leftarrow 1 + d_r\,(z_2 - z_{ref})$
3 $r_3 \leftarrow 1 + d_r\,(z_3 - z_{ref})$
4 $det \leftarrow \begin{vmatrix} r_1\,{z_1}^2 & z_1 & 1 \\ r_2\,{z_2}^2 & z_2 & 1 \\ r_3\,{z_3}^2 & z_3 & 1 \end{vmatrix}$
5 **si** $det < 10^{-8}$ **alors**
6 $\quad a \leftarrow 0 \quad b \leftarrow 0 \quad c \leftarrow 0$
7 **sinon**
8 $\quad det_a \leftarrow \begin{vmatrix} 1 & z_1 & x_1 \\ 1 & z_2 & x_2 \\ 1 & z_3 & x_3 \end{vmatrix}$
9 $\quad det_b \leftarrow \begin{vmatrix} r_1\,{z_1}^2 & 1 & x_1 \\ r_2\,{z_2}^2 & 1 & x_2 \\ r_3\,{z_3}^2 & 1 & x_3 \end{vmatrix}$
10 $\quad det_c \leftarrow \begin{vmatrix} r_1\,{z_1}^2 & z_1 & x_1 \\ r_2\,{z_2}^2 & z_2 & x_2 \\ r_3\,{z_3}^2 & z_3 & x_3 \end{vmatrix}$
11 $\quad a \leftarrow det_a / det$
12 $\quad b \leftarrow det_b / det$
13 $\quad c \leftarrow det_c / det$

Algorithme 3.2 `fitParabola`

global : z_{ref}, $D_r \in \mathbb{R}^3$
E/S : T // trace à ajuster
sortie : χ^2, $\max_{\chi^2}$, avg_d, $\max_d$ // propriétés de la trace

1 **répéter**
2 $\quad M \leftarrow \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad Y \leftarrow \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$
3 $\quad$ **pour** hit in T **faire**
4 $\quad\quad w \leftarrow hit.w$
5 $\quad\quad d_z \leftarrow hit.z_0 - z_{ref}$
6 $\quad\quad \delta \leftarrow {d_z}^2\ (1 + d_z\, T.d_r)$
7 $\quad\quad d \leftarrow \text{distance}(hit, T)$ // Distance du point à la paramétrisation de la trace
8 $\quad\quad d \leftarrow \text{distance}(hit, T)$ // Distance du point à la paramétrisation de la trace
9 $\quad\quad V \leftarrow \begin{pmatrix} 1 & d_z & \delta \end{pmatrix}^\intercal$
10 $\quad\quad M \leftarrow M + w\, V\, V^\intercal$ // M reste symétrique
11 $\quad\quad Y \leftarrow Y + w\, d\, V$
12 $\quad X \leftarrow \text{solve}(M, Y)$ // Résolution du système linéaire symétrique $M\,X = Y$
13 $\quad \begin{pmatrix} T.a_x \\ T.b_x \\ T.c_x \end{pmatrix} \leftarrow \begin{pmatrix} T.a_x \\ T.b_x \\ T.c_x \end{pmatrix} + X$
14 $\quad T.d_r \leftarrow \begin{pmatrix} 1 & T.d_r & {T.d_r}^2 \end{pmatrix} \cdot D_r$
15 **jusqu'à** $X_0 < 5 \cdot 10^{-3}$ *and* $X_1 < 5 \cdot 10^{-6}$ *and* $X_2 < 5 \cdot 10^{-9}$

16 // Calcul des propriétés de la trace T après ajustement
17 $\chi^2 \leftarrow 0 \qquad \max_{\chi^2} \leftarrow 0 \qquad \Sigma_d \leftarrow 0 \qquad \max_d \leftarrow 0$
18 **pour** hit in T **faire**
19 $\quad d \leftarrow \text{distance}(hit, T)$ // Distance du point à la paramétrisation de la trace
20 $\quad \chi_h^2 \leftarrow d^2\, hit.w$
21 $\quad \chi^2 \leftarrow \chi^2 + \chi_h^2$
22 $\quad \max_{\chi^2} \leftarrow \max\left(\max_{\chi^2}, \chi_h^2\right)$
23 $\quad \Sigma_d \leftarrow \Sigma_d + |d|$
24 $\quad \max_d \leftarrow \max\left(\max_d, |d|\right)$
25 $\text{avg}_d \leftarrow \Sigma_d / \text{size}(T)$

Les mesures de performances ont été faites sur une machine Intel de bureau avec un processeur `Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz`. Le temps est exprimé en cycles à la fréquence nominale et est divisé par le nombre d'éléments traités.

3.2 Étude sur la mesure du temps

Avant toute chose, il est crucial de comprendre comment mesurer le temps. Une étude préliminaire est nécessaire afin de s'assurer de la robustesse de nos méthodes.

3.2.1 Utilitaires de mesure du temps

Lorsque l'on souhaite mesurer le temps d'exécution d'un programme, il existe principalement deux méthodes :

— utiliser un outil externe tel que `perf` ou Vtune analysant l'état complet du programme à intervalle régulier,

— interroger l'horloge juste avant et juste après la fonction pour calculer la durée.

La première approche est simple à mettre en place, car elle ne requiert ni modification du programme, ni recompilation. Toutefois, la granularité n'est pas toujours suffisante pour mesurer le temps pris par des petites fonctions. De plus, une telle approche peut perturber les mesures de par l'exécution de code en parallèle.

La seconde approche ne souffre d'aucun de ces problèmes, mais nécessite de mesurer le temps depuis le programme lui-même. Cela permet aussi de synthétiser plus facilement les résultats en ayant les données et les mesures disponible en même temps. Pour toutes ces raisons, cette approche sera la seule utilisée ici.

Il existe plusieurs méthodes pour interroger l'horloge au sein du programme : les appels systèmes tels que `gettimeofday` et `clock_gettime` sous Linux, et les instructions accédant aux compteurs de performance du processeur tels que `rdtsc` et `rdpmc` en x86. Les langages de programmation proposent des interfaces standards pour accéder à l'horloge, utilisant généralement les appels systèmes.

Les appels systèmes permettent une grande portabilité : ils fonctionnent sur toutes les architectures, et ont des comportements complètement spécifiés. Un appel système prend entre 100 et 10000 cycles (avant les *patchs* contre *Spectre* [73] et *Meltdown* [74]). La granularité dépend de l'appel système et de la machine, mais ne reflète généralement pas la précision de l'horloge. Certaines horloge telles que celle utilisée par `gettimeofday` sont soumise à des synchronisations. Ces synchronisations peuvent se manifester par des sauts dans le temps dans le futur, mais aussi dans le passé. En prenant en considération ces propriétés, on en vient à choisir l'appel système suivant : `clock_gettime(CLOCK_MONOTONIC, ...)` [28, pp. 679–682]. Cet appel permet d'avoir une horloge précise qui n'est soumise à aucune synchronisation et est partagée entre tous les cœurs.

Les compteurs de performance (PMC) permettent un accès rapide et précis à certaines métriques du processeur relatives à la performance. La lecture de ces compteurs peut prendre quelques dizaines de cycles tout au plus. On retrouve ainsi sur la plupart des architectures un compteur pour le nombre d'instructions exécutées ou pour le nombre de cycles passés. L'architecture x86 donne accès à l'instruction non-privilégiée `rdtsc` qui permet de récupérer le nombre de cycles depuis un instant fixe dans le passé, généralement le démarrage de la machine. La sémantique de cette instruction a beaucoup varié durant l'histoire du x86 [75, chapitre 3.17.17]. La première version permettait de lire le nombre réel de cycles. Mais de par sa nature, cette instruction était fortement dépendante des changements de fréquences du processeur. Ainsi, les nouvelles versions ne comptent plus les cycles réels, mais des "cycles normalisés" à la fréquence nominale, indépendants des changements de fréquences. Les nouvelles versions de cette instruction permettent donc de mesurer le temps, l'ancien comportement est par ailleurs disponible avec l'instruction privilégiée `rdpmc`. De plus, cette horloge est également synchronisée entre tous les cœurs et tous les processeurs d'une même machine.

Des *flags* processeur permettent de déterminer le fonctionnement de cette instruction :
- `tsc` : l'instruction `rdtsc` est présente
- `constant_tsc` : `rdtsc` compte les cycles "normalisés"
- `nonstop_tsc` : les cycles continuent même lorsque le processeur est au repos
- `tsc_deadline_timer` : ce compteur est capable de gérer des interruptions régulières
- `tsc_adjust` : chaque *thread* matériel peut ajuster son propre compteur en imposant un décalage constant
- `rdtscp` : une variante de l'instruction est disponible avec des propriétés légèrement différentes
- ...

Pour avoir des mesures de temps fiables et précises, il est donc préférable d'utiliser l'instruction `rdtsc` lorsque celle-ci est disponible et à fréquence constante (*flags* : `tsc` + `constant_tsc`). Lorsque ce n'est pas le cas, on peut alors recourir à l'appel système `clock_gettime(CLOCK_MONOTONIC, ...)`, qui permet d'accéder à une horloge précise de

Listing 3.1 Mesure du temps fiable sous Linux

```
#include <time.h>

unsigned long long read_time_ns() {
  struct timespec_t t;
  clock_gettime(CLOCK_MONOTONIC, &t);
  return t.tv_nsec + 1000000000ull * t.tv_sec;
}
```

manière portable malgré un coût élevé (listing 3.1.

Les architectures ARM et Power ont des instructions proches, mais celles-ci sont privilégiées et non standardisées, ce qui rend difficile leur usage.

3.2.2 Distribution du temps de traitement

Afin d'avoir des résultats reproductibles, il est bon de mesurer le temps de traitement plusieurs fois (en répétant la fonction avec les mêmes données) et de synthétiser ces résultats. La méthode de synthèse des résultats dépend de la distribution du temps de traitement qu'il faut donc analyser.

L'analyse suivante porte sur le temps de traitement de `solveParabola`. Le temps de traitement est exprimé en cycles par élément et correspond au temps total mis par la fonction pour traiter N éléments en un seul *batch*, que l'on divise par le nombre d'éléments N afin d'avoir des résultats comparables avec des *batchs* de tailles différentes.

Pour un *batch* avec 23 254 éléments, on obtient la figure 3.1. On remarque que cette distribution n'est pas gaussienne : elle est constituée de plusieurs pics centrés en 32.87, 33.08 et 33.34. Le premier pic est le plus grand et représente à lui seul 75% de la distribution complète.

Si on zoom sur un pic, il est possible de le comparer avec une courbe gaussienne. Le premier pic de la distribution (figure 3.2) apparaît très proche d'une courbe gaussienne : l'erreur quadratique moyenne est de 0.408 sur ce pic.

En regardant l'évolution du temps de traitement au cours du temps lorsque les exécutions se suivent, on remarque l'émergence de motifs (figure 3.3). Régulièrement, une exécution de la fonction sera plus lente, la suivante n'étant pas affectée. On peut observer 3 périodes avec des impacts sur les performances différents : 1 ms, 8 ms et $\simeq 30$ ms.

La régularité de ces motifs nous indiquent que ce ne sont pas des évènements aléatoires. On remarque également que les ralentissements sont toujours uniquement sur une seule exécution et ne s'étendent jamais sur plusieurs. Ce phénomène est dû aux interruptions : celles-ci arrêtent l'exécution du code en cours, puis la relance une fois le traitement de l'interruption effectué. Durant l'interruption, le code est arrêté, mais le temps continue de s'écouler : d'où le ralentissement sur une unique exécution à chaque fois.

En ce qui concerne la régularité, le système dispose de plusieurs types d'interruptions. Certaines comme l'horloge noyau ou l'ordonnanceur de processus se produisent à intervalle régulier.

Si l'on regarde maintenant l'évolution de la distribution du temps de traitement en fonction de la taille du *batch* (figure 3.4), on peut voir que les pics supplémentaires deviennent de plus en plus haut et de plus en plus proche du premier pic à mesure que la taille du *batch* augmente. L'impact de l'interruption sur les performances diminue car le temps de

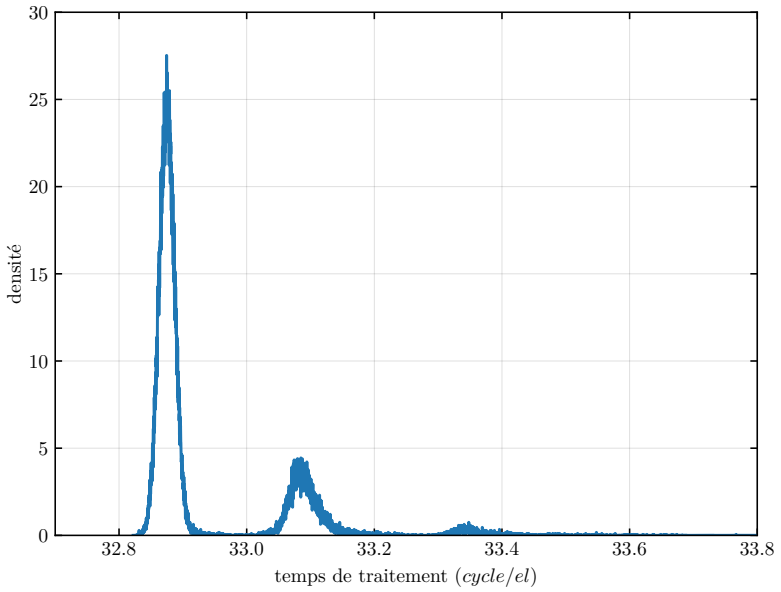


Figure 3.1 – Distribution du temps de traitement de `solveParabola` pour $N = 23\,254$

traitement de l'interruption devient négligeable face au temps d'exécution de la fonction en elle-même : les pics supplémentaires se rapprochent. Un *batch* plus grand implique également un temps d'exécution plus grand, donc une probabilité plus grande d'observer une interruption : les pics supplémentaires deviennent plus grands.

Il arrive un moment où le temps d'exécution total devient plus grand que la période d'une interruption : le pic correspondant à l'absence d'interruption disparaît au profit du pic suivant. Dans notre cas, cela arrive lorsque la taille de *batch* dépasse 100 000 éléments.

Au vu des distributions du temps de traitement que l'on observe, il nous faut une synthèse qui évite de prendre en compte les pics supplémentaires, ceux-ci ne reflétant pas les performances de la fonction. Les métriques classiques telles que la moyenne, la médiane ou le mode ne sont donc pas adaptées. La moyenne et la médiane sont influencées par les pics supplémentaires car ceux-ci les décalent. L'effet sera d'autant plus important que les pics supplémentaires seront grands. Le mode pourrait résoudre ces problèmes tant que les pics sont de tailles différentes. Or les deux premiers pics peuvent être de la même taille lorsque les interruptions affectent la moitié des exécutions. Dans ce cas, la mesure du mode sera imprécise et pourra "osciller" entre les deux pics ce qui donnerait une mesure "aléatoire".

Une façon de contourner ces problèmes serait de ne considérer que le premier pic, et d'appliquer ensuite une des métriques classiques sur ce pic. Toutefois, la détection effective des pics afin d'en isoler le premier est une tâche lourde qui demande beaucoup de valeurs :

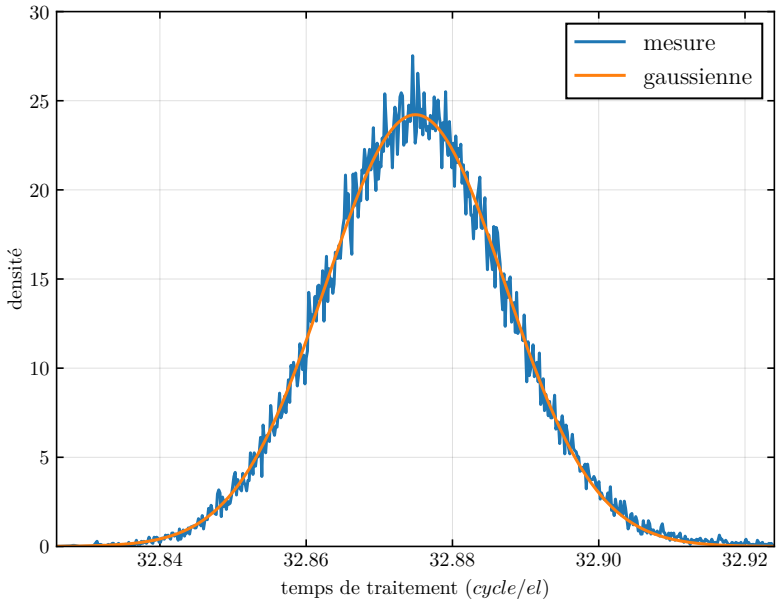


Figure 3.2 – Pic de la distribution du temps de traitement isolé (erreur quadratique moyenne : 0.408)

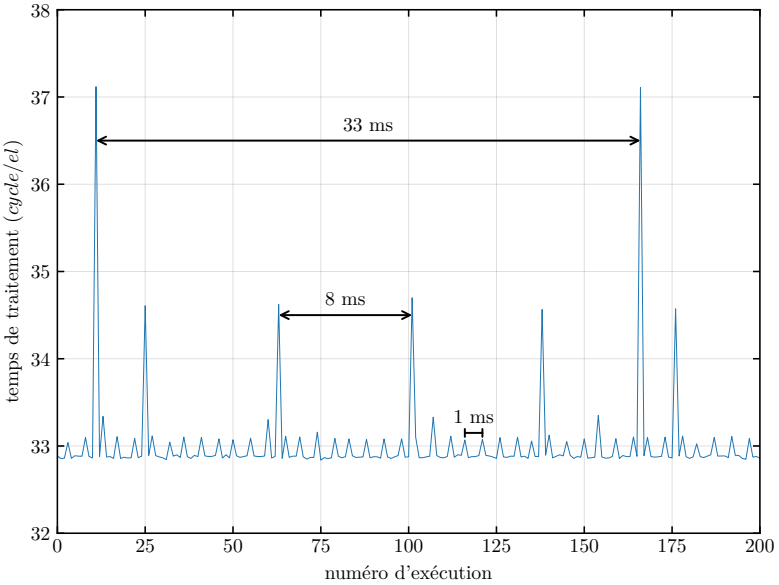


Figure 3.3 – Temps de traitement de `solveParabola` en fonction du temps (i7-4790)

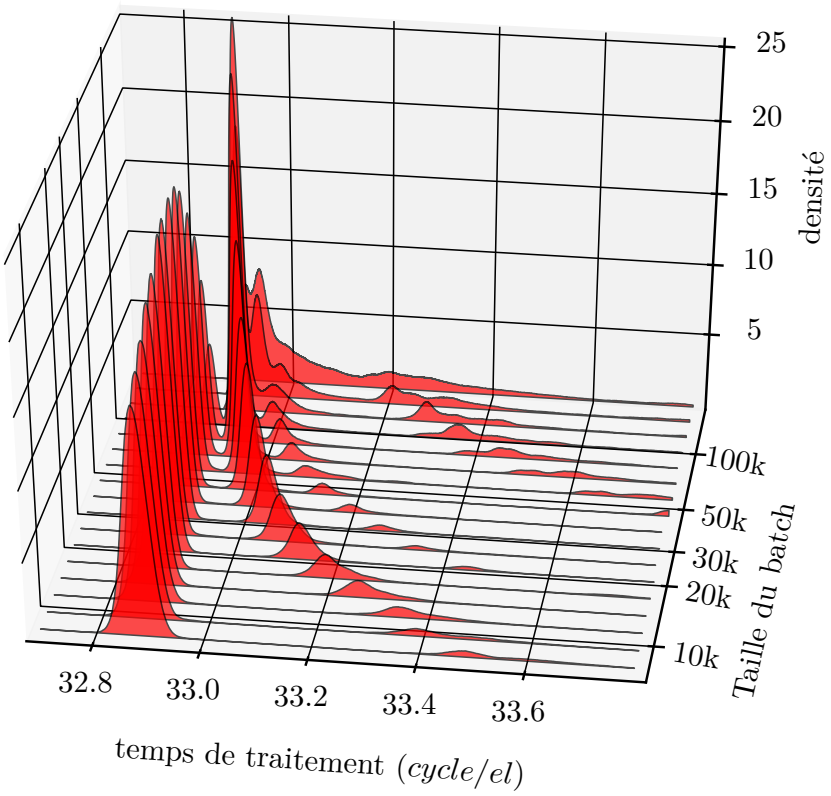


Figure 3.4 – Distribution du temps de traitement de `solveParabola` en fonction de la taille de *batch* (i7-4790)

les distributions ont été générées à partir de 100 000 exécutions pour chaque taille de *batch*. De plus, si deux pics sont suffisamment proches l'un de l'autre, ils peuvent se confondre ce qui rend leur séparation d'autant plus difficile.

En pratique la distribution d'un pic ne peut pas être parfaitement gaussienne : il existe une valeur minimum en deçà de laquelle la probabilité est strictement nulle. Le temps d'exécution est minoré. La synthèse retenue est donc de prendre le temps d'exécution minimum. Le minimum donne de bon résultats car la variance de chaque pic est faible, et correspond à une contrainte réelle sur le code à mesurer.

De plus, le minimum converge très vite et quelques dizaines de valeurs peuvent suffire pour avoir un résultat cohérent et reproductible. Enfin, le minimum peut se calculer à la volée sans avoir à stocker toutes les valeurs précédentes.

Dans le cas où le temps total mesuré est trop court par rapport à la granularité de l'horloge utilisée, il convient de mesurer plusieurs exécutions d'affilée. Cela aura pour conséquence d'augmenter le temps total mesuré, ainsi que de faire une moyenne sur un nombre faible de valeurs. Il sera dès lors possible de prendre le minimum de ces moyennes.

3.3 Analyse des performances

3.3.1 Comparaison des *memory layouts*

Les temps de traitement de `solveParabola` sur le processeur i7-4790 (tableau 3.1 et tableau 3.2) sont explicites : l'agencement mémoire *AoS* est plus lent, même lorsque le code n'est pas vectorisé, alors que *SoA* et *AoSoA* ont des performances similaires.

La vectorisation permet une accélération de $\times 2.9$ en *AoS*. Ce facteur est très bas par rapport au parallélisme du SIMD sur cette machine : 8 `float`s par registre. L'efficacité vectorielle est de 36%. Il faut d'avantage d'accès mémoires, bien que plus petits, et transposer les données afin de bénéficier de la vectorisation horizontale malgré l'agencement mémoire. Ceci ralentit considérablement le traitement des données.

La vectorisation est plus efficace en *SoA* et *AoSoA* atteignant une efficacité respective de 76% et 73% à agencement mémoire égal. La version SIMD est meilleure avec une efficacité de 97%. Celle-ci implémente la condition (algorithme 3.1, ligne 5) avec une affectation masquée uniquement, tandis que la version vectorisée a recours à un branchement afin d'éviter cette affectation masquée lorsque tous les éléments du vecteur vérifie la condition. Un tel test requiert de déplacer les données d'un registre SIMD entre les lignes. Si le temps de traitement est mesuré en ayant enlevé au préalable la condition, les versions vectorisée et SIMD sont proches : respectivement 4.06 cycle/el et 3.80 cycle/el. L'impact sur les performances de ce branchement est important de par la nature des données : la condition n'est vraie qu'occasionnellement.

Table 3.1 – Temps de traitement de `solveParabola` de 100 000 éléments en cycle/élément (i7-4790)

Scalaire			Vectoriel			SIMD
AoS	*SoA*	*AoSoA*	*AoS*	*SoA*	*AoSoA*	*SoA*
32.94	30.94	31.06	11.50	5.08	5.33	4.01

Table 3.2 – Accélérations du traitement de `solveParabola` pour 100 000 éléments (i7-4790)

Accélérations		Scalaire			Vectoriel			SIMD
		AoS	*SoA*	*AoSoA*	*AoS*	*SoA*	*AoSoA*	*SoA*
Scalaire	*AoS*	1						
	SoA	1.06	1					
	AoSoA	1.06	1.00	1				
Vectoriel	*AoS*	2.86	2.69	2.70	1			
	SoA	6.49	6.09	6.12	2.26	1		
	AoSoA	6.18	5.80	5.83	2.16	0.95	1	
SIMD	*SoA*	8.22	7.72	7.75	2.87	1.27	1.33	1

3.3.2 Influence de la taille des données

Afin de continuer l'analyse des performances, il nous faut regarder l'impact de la taille du *batch* : figure 3.5. Les courbes apparaissent assez plates : le temps de traitement dépend peu de la taille du *batch*. On observe toutefois un motif classique en dent de scie pour de petits *batchs*, ainsi qu'un ralentissement pour les gros *batchs*.

Ce dernier ralentissement correspond à la sortie du cache L3 : il y a trop de données pour que celles-ci soient dans le cache. Les données doivent donc être chargées depuis la mémoire externe (RAM). Le débit de cette mémoire étant plus faible que le cache, le traitement des données devient plus lent. Plus la version de la fonction était rapide lorsque les données sont en cache, plus l'impact sera important : il faut un débit plus haut pour maintenir la grande vitesse de traitement.

Il n'y a cependant pas de tel ralentissement pour les sorties des caches L1 et L2. Cela signifie que le débit du L3 est suffisant pour traiter les données à pleine vitesse. L'explication est simple : `solveParabola` nécessite d'effectuer beaucoup plus d'opérations arithmétiques que d'accès mémoires : l'Intensité Arithmétique de cette fonction est de 7.2. Ce problème est donc principalement *compute bound*.

Le motif en dents de scie n'apparaît, quant à lui, que pour les versions vectorisées et SIMD traitant de petits *batchs* (figure 3.6). La période de ce motif est de 8 qui correspond à la taille des registres AVX : si le nombre d'éléments n'est pas un multiple de 8, il est nécessaire de traiter les éléments restants différemment. Une méthode simple pour traiter ces

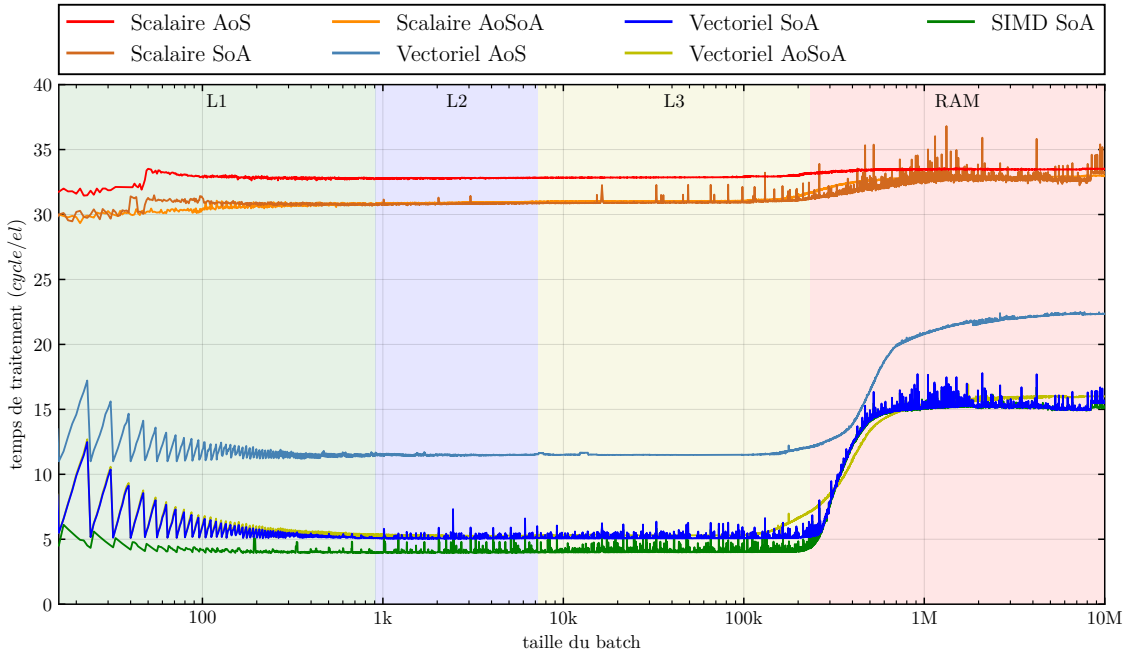


Figure 3.5 – Temps de traitement de `solveParabola` en fonction de la taille du *batch*

éléments est d'utiliser une boucle scalaire ayant au plus 7 itérations. Celle-ci est d'ailleurs utilisée par les compilateurs.

Ainsi, si on ajoute un élément à traiter, la boucle scalaire aura une itération de plus. Mais si un multiple de 8 est atteint, les 7 itérations scalaires seront remplacées par une seule itération vectorielle. Cette itération vectorielle étant bien plus rapide que les 7 itérations scalaires, le temps total diminue. D'où le motif en dents de scie.

La version SIMD implémente le traitement du reste de manière différente : les derniers éléments sont placés dans un registre SIMD (qui sera donc partiellement vide). Les calculs ainsi effectués sont strictement identiques aux itérations complètes, mais tous les accès mémoires sont masqués : seuls les éléments traités sont accédés. Ainsi, qu'il y ait 1 ou 7 éléments restants, le temps d'exécution total sera le même, d'où la vitesse de traitement par élément qui augmente lorsque l'on ajoute des éléments, jusqu'au prochain multiple de 8 où une nouvelle itération vectorielle sera requise.

Dans les 2 cas, la taille des dents diminue avec le nombre d'éléments à traiter : le temps de calcul des éléments restants devient négligeable par rapport au temps pour traiter la boucle complète. La performance étant le temps total divisé par le nombre d'éléments, la contribution des éléments restants tend vers 0.

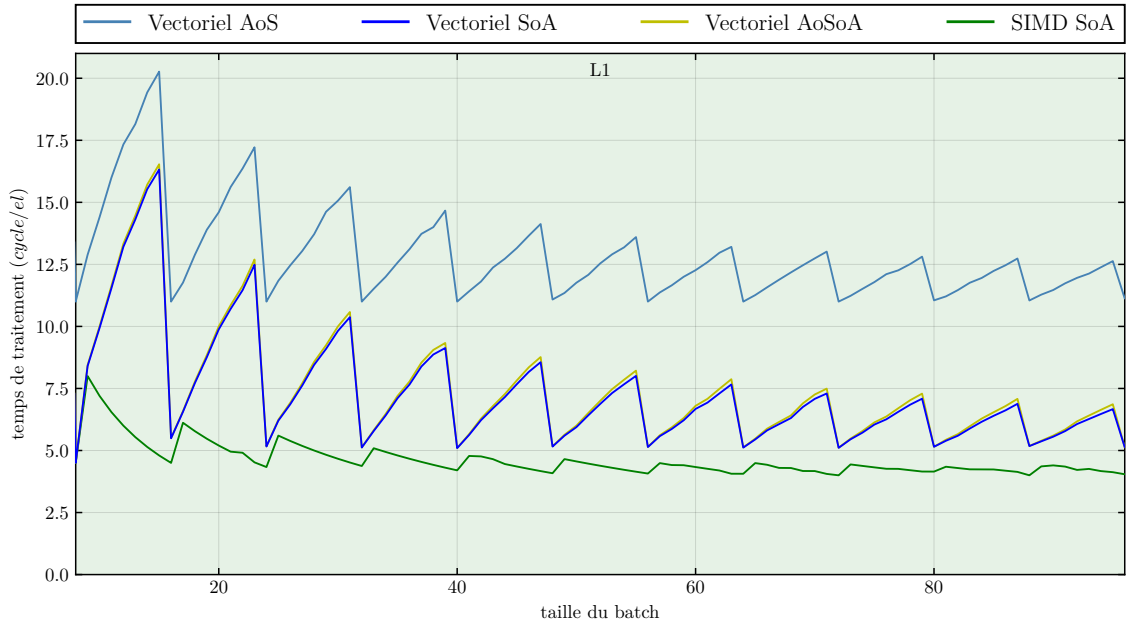


Figure 3.6 – Temps de traitement de `solveParabola` en fonction de la taille du *batch* : impact de la taille des registres

Les courbes de performance des versions *SoA* vectorisée et SIMD font apparaître des pics : figure 3.7. Ces pics ne sont pas des artefacts de mesure : ils sont dus aux évictions systématiques de cache, lorsque la taille du *batch* est un multiple d'une grande puissance de 2. Le pic au centre de la figure se situe à $N = 16384 = 2^{14}$. En *SoA*, la fonction `solveParabola` nécessite en effet 10 références actives, alors que les caches L1 et L2 de la machine sont 8-associatifs.

Par ailleurs, la version *AoSoA* n'est pas affectée par ce phénomène. Ainsi, le temps de traitement de cette version reste constant, même au voisinage des puissances de 2.

3.3.3 Impact de la précision

La précision de calcul influence la vitesse de traitement (figure 3.8). Ainsi le code en `float` est en moyenne 2 fois plus rapide que le code en `double`.

On remarque cependant que durant la sortie de cache la version `float` parvient à être 6.4 fois plus rapide pour traiter 220 000 éléments. Un `double` occupant 2 fois plus de place, la version les utilisant remplis le cache 2 fois plus vite : la sortie de cache arrive plus tôt. Ainsi, la version `double` commence à sortir du cache pour $\simeq$ 150 000 éléments, tandis que la version `float` sort du cache à partir de $\simeq$ 300 000 éléments. Entre ces deux valeurs, la

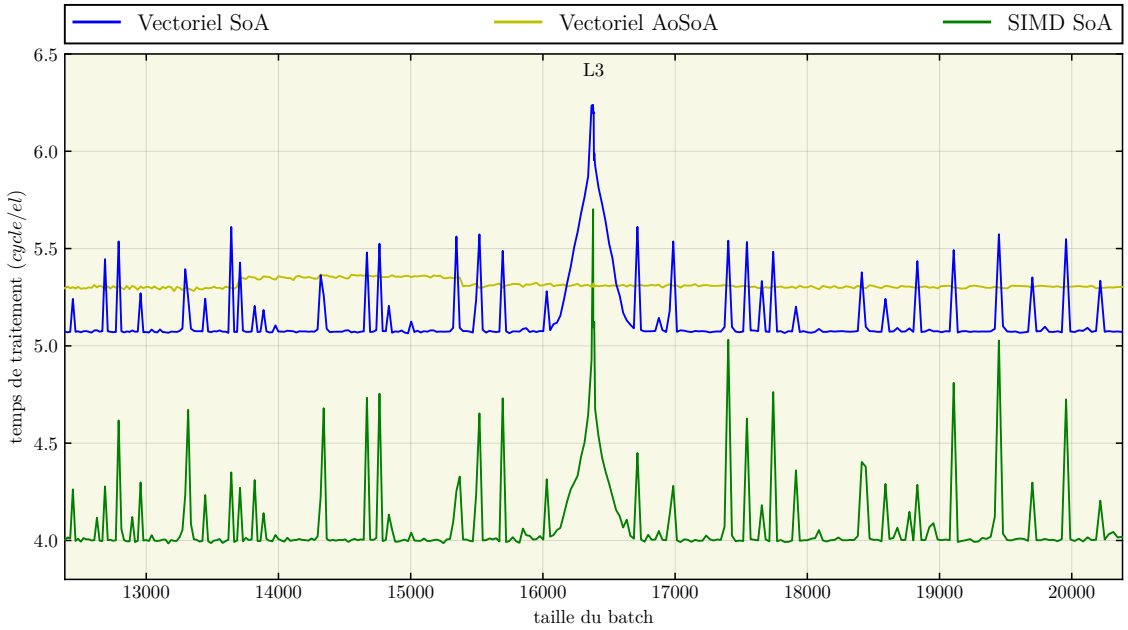


Figure 3.7 – Temps de traitement de `solveParabola` en fonction de la taille du *batch* : évictions systématiques de cache

version `float` est beaucoup plus rapide.

On remarque également qu'avant la sortie de cache, l'accélération est légèrement supérieure à 2 atteignant $\times 2.14$. Ceci s'explique par la division : en `double`, celle-ci est plus lente (27 cycles contre 13 cycles en simple précision) et traite 2 fois moins d'éléments.

Les pics correspondant aux évictions systématiques de cache sont plus hauts : un défaut de cache coûte plus cher en `double` de par la taille des données.

3.3.4 *Multithreading*

Le *multithreading* est géré avec OpenMP. Cela permet une implémentation à la fois simple et efficace. Les performances avec OpenMP sont reportées dans le tableau 3.3. L'efficacité parallèle est définie comme l'accélération due au *multithreading* divisée par le nombre de cœurs utilisés.

Pour traiter 100 000 éléments, l'exécution sur les 4 cœurs de la machine permet de réduire le temps de calcul d'un facteur équivalent : l'efficacité obtenue est supérieure à 90%. Ce code passe bien à l'échelle lorsque les données sont en cache : le cache L3 a une bande passante suffisamment élevée pour alimenter les 4 cœurs de la machine.

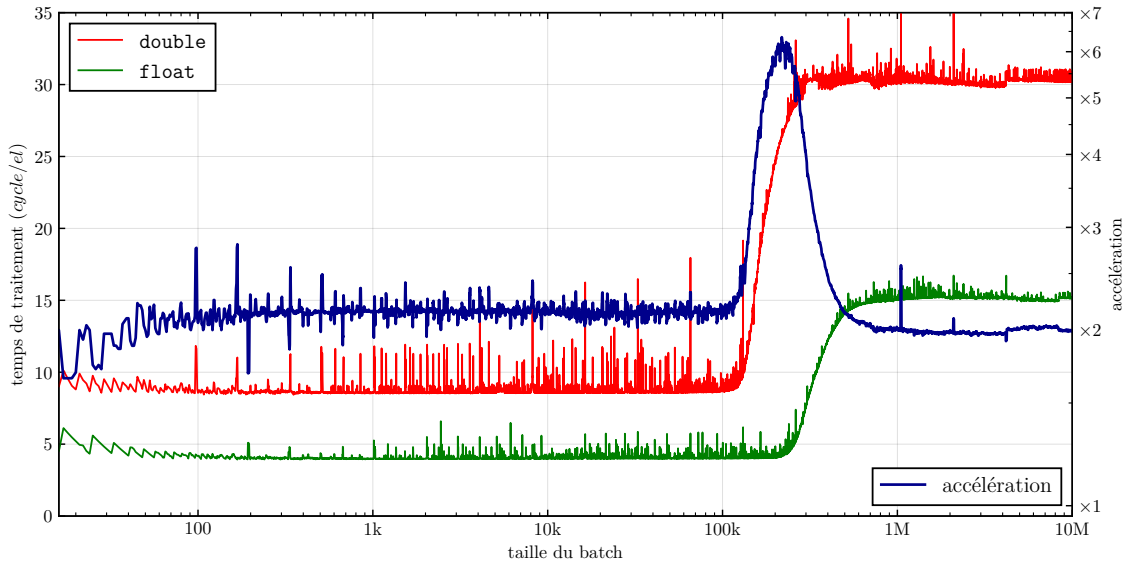


Figure 3.8 – Accélération dû à la précision : `solveParabola` SIMD

Le SMT [76] (*Simultaneous Multithreading*) permet d'améliorer l'efficacité d'environ 10%. Le gain est limité par la forte utilisation des unités fonctionnelles : il n'y a pas assez d'unités pour que les 2 *threads* puissent s'exécuter à pleine vitesse.

La performance avec OpenMP dépend aussi de la taille du *batch* (figure 3.9). Avec peu d'éléments à traiter, la vitesse de traitement est basse. En effet, le *multithreading* a un coût intrinsèque constant dû à la gestion des *threads* (création, synchronisation). Ce coût est prépondérant pour de petites données, et devient négligeable lorsque le temps d'exécution total devient plus long. L'exécution en *SoA* sur les 4 cœurs devient plus rapide que l'exécution sur un seul cœur à partir de $\simeq 700$ éléments. Avant ce point, le coût du *multithreading* est trop grand.

La sortie du cache L3 est plus importante qu'avec un seul cœur : la bande passante de la mémoire externe est partagée entre tous les cœurs. Après la sortie de cache, toutes les versions *SoA* et *AoSoA* vont à la même vitesse. Le code est limité uniquement par la bande passante mémoire : il est *memory bound*. On remarque d'ailleurs que le temps de traitement limite avec 4 cœurs correspond au temps de traitement atteint avec 1 cœur. Le code ne passe plus à l'échelle lorsque les données ne sont pas en cache.

Les versions *AoS* présentent le même comportement avec une vitesse de traitement moindre : ces implémentations ont besoin de plus de bande passante.

Table 3.3 – Accélération de `solveParabola` dû au *multithreading* (i7-4790)

$N = 100\,000$		temps de traitement (cycle/el)			accélération depuis 1 cœur		efficacité parallèle	
		1 cœur	4 cœurs	8 *threads*	4 cœurs	8 *threads*	4 cœurs	8 *threads*
Scalaire	*AoS*	32.94	8.25	7.62	3.99	4.32	99.8%	108.1%
	SoA	30.94	7.75	7.04	3.99	4.39	99.8%	109.9%
	AoSoA	31.06	7.78	6.96	3.99	4.46	99.8%	111.5%
Vectoriel	*AoS*	11.50	2.93	2.63	3.93	4.38	98.2%	109.4%
	SoA	5.08	1.31	1.21	3.89	4.21	97.2%	105.2%
	AoSoA	5.33	1.36	1.22	3.92	4.38	97.9%	109.5%
SIMD	*SoA*	4.01	1.08	0.99	3.73	4.06	93.2%	101.5%

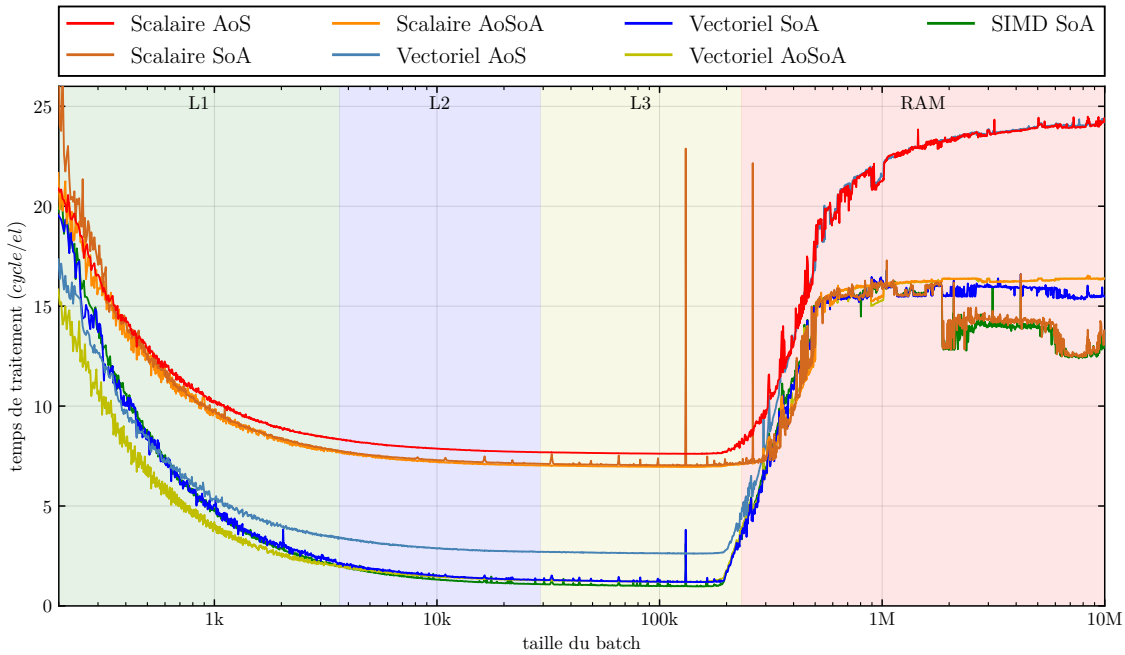


Figure 3.9 – Temps de traitement de `solveParabola` en fonction de la taille du *batch* avec OpenMP

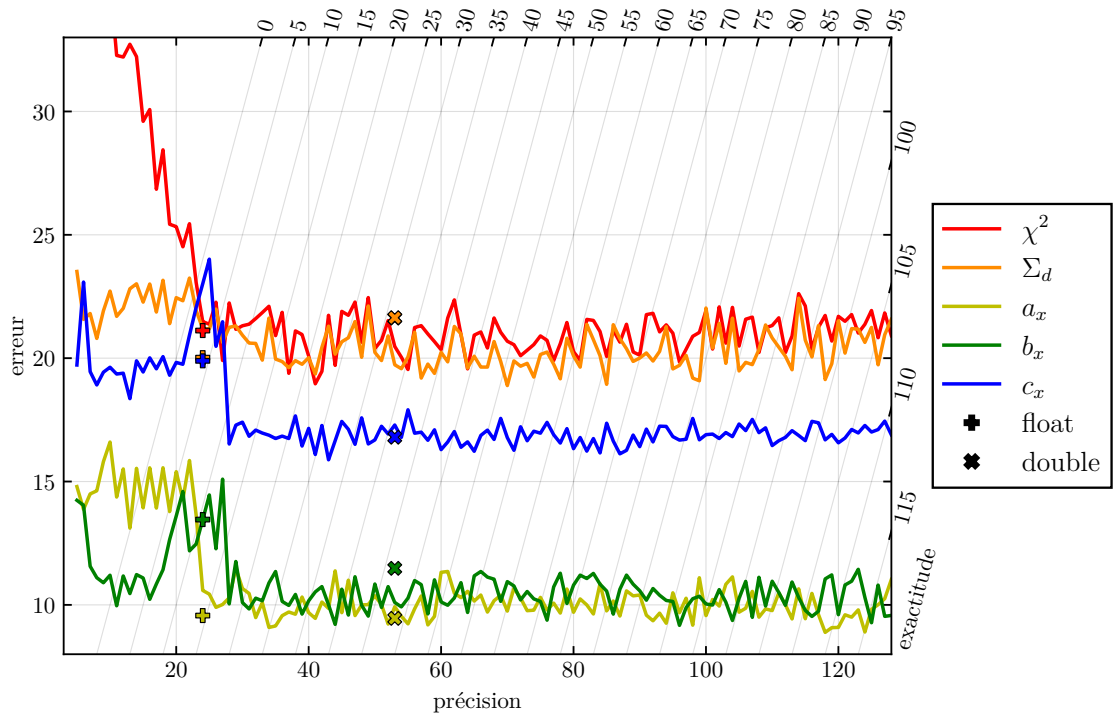


Figure 3.10 – Erreur de `fitParabola` en fonction de la précision en bits

3.4 Analyse de la précision

La figure 3.10 reporte l'erreur maximum observée sur la sortie de `fitParabola` en fonction de la précision de calcul. Pour une précision > 30 bits, l'erreur est constante. Pour de plus faibles précisions, les résultats sont instables et l'erreur augmente : les résultats temporaires souffrent de cancellations catastrophiques. L'exactitude baisse donc plus vite, mais arrive à rester positive malgré ces instabilités pour une précision de 24 bits.

On remarque que les différentes grandeurs en sortie n'ont pas la même erreur : a_x et b_x ont une erreur moyenne de 10 bits, tandis que χ^2 et Σ_d ont une erreur moyenne de 20 bits. Ces dernières étant des sommes, elles sont plus sensibles aux instabilités numériques (absorptions et cancellations).

Enfin, les versions `float` et `double` ont des erreurs différentes des versions MPFR à 24 bits et 54 bits de précision respectivement. Le code a été compilé par icc avec l'option par défaut : `-fp-model fast`. Cette option relâche les contraintes sur le calcul flottant afin d'optimiser d'avantage le programme. Les résultats ne sont pas plus mauvais, ils sont

différents. Dans certains cas, ils peuvent même être plus exacts comme la valeur de c_x en `float`. Il se peut cependant que ces résultats plus exacts ne soit que le fruit du hasard où les bits perdus (et donc aléatoires) prennent une valeur proche de la valeur réelle [77].

3.5 synthèse

Nous avons vu dans cette partie les 3 points suivants :

— une méthode pour mesurer le temps d'une fonction de par l'analyse de la distribution du temps,

— les effets des différentes transformations et de détails architecturaux sur la vitesse de traitement d'une fonction,

— les effets de la précision utilisée lors des calculs sur l'exactitude des résultats.

Ceci nous permettra une analyse plus rapide et concise des résultats futures.

Chapitre 4

Factorisation de Cholesky

Cet algorithme est utilisé à plusieurs endroits au sein de LHCb avec des matrices allant de 2×2 à 5×5. En particulier, il est utilisé dans le filtre de Kalman (chapitre 5), ce qui en fait un algorithme de choix à accélerer. La petite taille des matrices à traiter rend l'utilisation de bibliothèques existantes inefficace.

Ce chapitre présente les optimisations apportées à la factorisation de Cholesky et leur impact sur la vitesse de traitement de la factorisation. Notre implémentation sera également comparée avec les bibliothèques d'algèbre linéaire existantes.

4.1 Algorithme

La factorisation de Cholesky [78, 79] (aussi appelée décomposition de Cholesky) est un algorithme d'algèbre linéaire qui permet d'exprimer une matrice symétrique définie positive comme le produit d'une matrice triangulaire par sa transposée : $A = L \cdot L^{\intercal}$. Elle peut être utilisée pour résoudre des systèmes linéaires dont les contraintes physiques imposent la symétrie du système. Elle est ainsi utilisée pour le *matching* par covariance et l'analyse de scènes mobiles [80] et l'étiquetage pour segmenter des scènes fixes [81].

La résolution complète d'un système à l'aide de la factorisation de Cholesky comporte 3 étapes : la factorisation, la descente et la remontée. La descente et la remontée sont groupées en une seule étape : la résolution.

4.1.1 Factorisation

La factorisation de Cholesky d'une matrice $n \times n$ a une compléxité algorithmique de $n^3/3$. C'est moitié moins d'opérations que la décomposition LU ($2n^3/3$). La factorisation de Cholesky est aussi numériquement plus stable que cette dernière [82, 83].

Algorithme 4.1 Factorisation de Cholesky (`factorize`)

entrée : A // matrice $n \times n$ symétrique définie positive
sortie : L // matrice $n \times n$ triangulaire inférieure

1 **pour** $j = 0 : n-1$ **faire**
2 $\quad s \leftarrow A(j,j)$
3 $\quad$ **pour** $k = 0 : j-1$ **faire**
4 $\quad\quad s \leftarrow s - L(j,k)^2$
5 $\quad L(j,j) \leftarrow \sqrt{s}$
6 $\quad$ **pour** $i = j+1 : n-1$ **faire**
7 $\quad\quad s \leftarrow A(i,j)$
8 $\quad\quad$ **pour** $k = 0 : j-1$ **faire**
9 $\quad\quad\quad s \leftarrow s - L(i,k) \cdot L(j,k)$
10 $\quad\quad L(i,j) \leftarrow s / L(j,j)$

Cet algorithme (algorithme 4.1) est naturellement en-place du fait que chaque élément de la matrice d'origine n'est accédé qu'une seule fois et ce avant que l'élément correspondant ne soit écrit. Il est donc possible que L et A partagent le même emplacement mémoire si A n'est plus utilisé par la suite.

La factorisation d'une matrice $n \times n$ nécessite n racines carrées et $(n^2 + 3n)/2$ divisions qui sont des opérations lentes, particulièrement en double précision.

4.1.2 Résolution

Une fois la forme factorisée de A obtenue, il est possible de résoudre simplement le système $A \cdot X = R$. En effet, comme $A = L \cdot L^{\intercal}$, le système à résoudre devient $L \cdot L^{\intercal} \cdot X = R$, qui se décompose ainsi : $L \cdot Y = R$ et $L^{\intercal} \cdot X = Y$. Ces deux systèmes triangulaires sont facilement résolubles grâces aux algorithmes de descente et de remontée (algorithme 4.2).

Tout comme la factorisation, la résolution est naturellement en-place : R, Y et X peuvent partager le même emplacement mémoire.

4.2 Transformations

L'approche utilisée pour optimiser Cholesky est d'appliquer différentes transformations et d'évaluer leur produit cartésien. On a ainsi une recherche exhaustive des meilleures combinaisons de transformations. Cette approche est également utilisée par Spiral [84] et Atlas [85].

Algorithme 4.2 Résolution (`substitute`)

entrée : L // matrice $n \times n$ triangulaire inférieure
entrée : R // vecteur de taille n
sortie : X // vecteur de taille n, solution de $L \cdot L^T \cdot X = R$
temp : Y // vecteur de taille n
// Descente
1 **pour** $i = 0 : n-1$ **faire**
2 $\quad s \leftarrow R(i)$
3 $\quad$ **pour** $j = 0 : i-1$ **faire**
4 $\quad\quad s \leftarrow s - L(i,j) \cdot Y(j)$
5 $\quad Y(i) \leftarrow s / L(i,i)$
// Remontée
6 **pour** $i = n-1 : 0$ **faire**
7 $\quad s \leftarrow Y(i)$
8 $\quad$ **pour** $j = i+1 : n-1$ **faire**
9 $\quad\quad s \leftarrow s - L(j,i) \cdot X(j)$
10 $\quad X(i) \leftarrow s / L(i,i)$

4.2.1 Traitement par lot

Lorsque l'on travaille avec des petites matrices, il n'y a pas de dimension suffisament grande pour que la parallélisation le long de cette dimension soit efficace. Par exemple, une boucle avec seulement 3 itérations ne pourra pas être vectorisée efficacement.

Pour contourner ce problème, on va donc regrouper les matrices afin de calculer leur factorisation en un seul appel : on traite les matrices par lot (*batch* en anglais). Puisque l'on a besoin de traiter beaucoup de matrices, on a donc à disposition suffisament de parallélisme pour que la vectorisation et le *multithreading* puissent être efficace. Cette approche est aussi utilisé par Jack Dongarra et son équipe pour la factorisation LU [86] et la factorisation de Cholesky sur GPU [87].

Algorithme 4.3 Traitement par lot

entrée : A // liste de N matrices $n \times n$ symétriques définies positives
entrée : R // liste de N vecteurs de taille n
sortie : X // liste de N vecteurs de taille n
1 **pour** $u = 0 : N-1$ **faire**
2 $\quad L_u \leftarrow \text{factorize}(A_u)$
3 $\quad X_u \leftarrow \text{substitute}(L_u, R_u)$

Table 4.1 – Nombre d'opérations flottantes de Cholesky

(a) Versions classiques

Algorithme	flop	accès mémoire	IA
factorisation	$\frac{1}{6}\left(2n^3+3n^2+7n\right)$	$\frac{1}{6}\left(2n^3+16n\right)$	$\simeq 1$
résolution	$2n^2$	$2n^2+4n$	$\simeq 1$
résolution complète	$\frac{1}{6}\left(2n^3+15n^2+7n\right)$	$\frac{1}{6}\left(2n^3+12n^2+40n\right)$	$\simeq 1$

(b) Versions déroulées et scalarisées

Algorithme	flop	accès mémoire	IA
factorisation	$\frac{1}{6}\left(2n^3+3n^2+7n\right)$	$\frac{1}{2}\left(2n^2+5n\right)$	$\simeq 0.33n$
résolution	$2n^2$	$\frac{1}{2}\left(n^2+5n\right)$	$\simeq 4$
résolution complète	$\frac{1}{6}\left(2n^3+15n^2+7n\right)$	$\frac{1}{2}\left(n^2+6n\right)$	$\simeq 0.66n$

4.2.2 Déroulage et scalarisation

Les matrices à factoriser sont petites et leur taille est connue à l'avance, on peut donc dérouler complètement la factorisation et la résolution et ainsi scalariser le code. On passe ainsi des algorithmes non-déroulés 4.1 et 4.2 aux algorithmes déroulés et scalarisés 4.4 et 4.5.

Cette transformation permet d'augmenter l'Intensité Arithmétique en s'affranchissant des accès mémoires redondants. Il est possible de l'augmenter davantage en fusionnant les deux fonctions et ainsi de s'affranchir des accès mémoires de L : l'enregistrement de L (algorithme 4.4, lignes 21–30) et le chargement de L (algorithme 4.5, lignes 1–10) ne sont plus nécessaires du fait que les registres contenant L restent disponibles pour la résolution.

On réduit ainsi le nombre total d'accès mémoires (tableau 4.1b) et on obtient l'algorithme 4.6 . Les accès mémoire restant sont le chargement de la matrice A et du vecteur R, ainsi que l'enregistrement du vecteur X. Tous les autres accès mémoire, y compris L, ont été éliminés. Du *spill code* peut toutefois apparaître si la matrice est trop grande.

Algorithme 4.4 Factorisation de Cholesky de matrices 4×4 déroulée et scalarisée

entrée : A // matrice 4×4 symétrique
sortie : L // matrice 4×4 triangulaire

// Charge A en registre
1 $a_{00} \leftarrow A(0,0)$
2 $a_{10} \leftarrow A(1,0)$
3 $a_{11} \leftarrow A(1,1)$
4 $a_{20} \leftarrow A(2,0)$
5 $a_{21} \leftarrow A(2,1)$
6 $a_{22} \leftarrow A(2,2)$
7 $a_{30} \leftarrow A(3,0)$
8 $a_{31} \leftarrow A(3,1)$
9 $a_{32} \leftarrow A(3,2)$
10 $a_{33} \leftarrow A(3,3)$

// Factorise A
11 $l_{00} \leftarrow \sqrt{a_{00}}$
12 $l_{10} \leftarrow a_{10}/l_{00}$
13 $l_{20} \leftarrow a_{20}/l_{00}$
14 $l_{30} \leftarrow a_{30}/l_{00}$
15 $l_{11} \leftarrow \sqrt{a_{11} - {l_{10}}^2}$
16 $l_{21} \leftarrow (a_{21} - l_{20} \cdot l_{10}) / l_{11}$
17 $l_{31} \leftarrow (a_{31} - l_{30} \cdot l_{10}) / l_{11}$
18 $l_{22} \leftarrow \sqrt{a_{22} - {l_{20}}^2 - {l_{21}}^2}$
19 $l_{32} \leftarrow (a_{32} - l_{30} \cdot l_{20} - l_{31} \cdot l_{21}) / l_{22}$
20 $l_{33} \leftarrow \sqrt{a_{33} - {l_{30}}^2 - {l_{31}}^2 - {l_{32}}^2}$

// Enregistre L en mémoire
21 $L(0,0) \leftarrow l_{00}$
22 $L(1,0) \leftarrow l_{10}$
23 $L(1,1) \leftarrow l_{11}$
24 $L(2,0) \leftarrow l_{20}$
25 $L(2,1) \leftarrow l_{21}$
26 $L(2,2) \leftarrow l_{22}$
27 $L(3,0) \leftarrow l_{30}$
28 $L(3,1) \leftarrow l_{31}$
29 $L(3,2) \leftarrow l_{32}$
30 $L(3,3) \leftarrow l_{33}$

Algorithme 4.5 Résolution de matrices 4×4 factorisées, déroulée et scalarisée

entrée : L // matrice 4×4 triangulaire
entrée : R // vecteur de taille 4
sortie : X // solution de $L \cdot L^T \cdot X = R$

// Charge L en registre
1 $l_{00} \leftarrow L(0,0)$
2 $l_{10} \leftarrow L(1,0)$
3 $l_{11} \leftarrow L(1,1)$
4 $l_{20} \leftarrow L(2,0)$
5 $l_{21} \leftarrow L(2,1)$
6 $l_{22} \leftarrow L(2,2)$
7 $l_{30} \leftarrow L(3,0)$
8 $l_{31} \leftarrow L(3,1)$
9 $l_{32} \leftarrow L(3,2)$
10 $l_{33} \leftarrow L(3,3)$

// Charge R en registre
11 $r_0 \leftarrow R(0)$
12 $r_1 \leftarrow R(1)$
13 $r_2 \leftarrow R(2)$
14 $r_3 \leftarrow R(3)$

// Descente
15 $y_0 \leftarrow r_0/l_{00}$
16 $y_1 \leftarrow (r_1 - l_{10} \cdot y_0) / l_{11}$
17 $y_2 \leftarrow (r_2 - l_{20} \cdot y_0 - l_{21} \cdot y_1) / l_{22}$
18 $y_3 \leftarrow (r_3 - l_{30} \cdot y_0 - l_{31} \cdot y_1 - l_{32} \cdot y_1) / l_{33}$

// Remontée
19 $x_3 \leftarrow y_3/l_{33}$
20 $x_2 \leftarrow (y_2 - l_{32} \cdot x_3) / l_{22}$
21 $x_1 \leftarrow (y_1 - l_{21} \cdot x_2 - l_{31} \cdot x_3) / l_{11}$
22 $x_0 \leftarrow (y_0 - l_{10} \cdot x_1 - l_{20} \cdot x_2 - l_{30} \cdot x_3) / l_{00}$

// Enregistre X en mémoire
23 $X(3) \leftarrow x_3$
24 $X(2) \leftarrow x_2$
25 $X(1) \leftarrow x_1$
26 $X(0) \leftarrow x_0$

Algorithme 4.6 Résolution complète avec la factorisation de Cholesky pour des matrices 4×4, déroulée et scalarisée

entrée : A // matrice 4×4 symétrique définie positive
entrée : R // vecteur de taille 4
sortie : X // vecteur de taille 4, solution de $L \cdot L^T \cdot X = R$

// Charge A en registre
1 $a_{00} \leftarrow A(0,0)$
2 $a_{10} \leftarrow A(1,0) \quad a_{11} \leftarrow A(1,1)$
3 $a_{20} \leftarrow A(2,0) \quad a_{21} \leftarrow A(2,1) \quad a_{22} \leftarrow A(2,2)$
4 $a_{30} \leftarrow A(3,0) \quad a_{31} \leftarrow A(3,1) \quad a_{32} \leftarrow A(3,2) \quad a_{33} \leftarrow A(3,3)$

// Charge R en registre
5 $r_0 \leftarrow R(0) \quad r_1 \leftarrow R(1) \quad r_2 \leftarrow R(2) \quad r_3 \leftarrow R(3)$

// Factorise A
6 $l_{00} \leftarrow \sqrt{a_{00}}$
7 $l_{10} \leftarrow a_{10}/l_{00}$
8 $l_{20} \leftarrow a_{20}/l_{00}$
9 $l_{30} \leftarrow a_{30}/l_{00}$
10 $l_{11} \leftarrow \sqrt{a_{11} - {l_{10}}^2}$
11 $l_{21} \leftarrow (a_{21} - l_{20} \cdot l_{10}) / l_{11}$
12 $l_{31} \leftarrow (a_{31} - l_{30} \cdot l_{10}) / l_{11}$
13 $l_{22} \leftarrow \sqrt{a_{22} - {l_{20}}^2 - {l_{21}}^2}$
14 $l_{32} \leftarrow (a_{32} - l_{30} \cdot l_{20} - l_{31} \cdot l_{21}) / l_{22}$
15 $l_{33} \leftarrow \sqrt{a_{33} - {l_{30}}^2 - {l_{31}}^2 - {l_{32}}^2}$

// Descente
16 $y_0 \leftarrow r_0/l_{00}$
17 $y_1 \leftarrow (r_1 - l_{10} \cdot y_0) / l_{11}$
18 $y_2 \leftarrow (r_2 - l_{20} \cdot y_0 - l_{21} \cdot y_1) / l_{22}$
19 $y_3 \leftarrow (r_3 - l_{30} \cdot y_0 - l_{31} \cdot y_1 - l_{32} \cdot y_1) / l_{33}$

// Remontée
20 $x_3 \leftarrow y_3/l_{33}$
21 $x_2 \leftarrow (y_2 - l_{32} \cdot x_3) / l_{22}$
22 $x_1 \leftarrow (y_1 - l_{21} \cdot x_2 - l_{31} \cdot x_3) / l_{11}$
23 $x_0 \leftarrow (y_0 - l_{10} \cdot x_1 - l_{20} \cdot x_2 - l_{30} \cdot x_3) / l_{00}$

// Enregistre X en mémoire
24 $X(3) \leftarrow x_3 \quad X(2) \leftarrow x_2 \quad X(1) \leftarrow x_1 \quad X(0) \leftarrow x_0$

4.2.3 Racine carrée inverse

La factorisation de Cholesky d'une matrice $n \times n$ nécessite le calcul de n racines carrées et de $(n^2 + 3n)/2$ divisions. Ces divisions faisant toujours intervenir les racines carrées au dénominateur, il est possible d'économiser la majorité des divisions en calculant l'inverse des racines carrées, puis en remplaçant les divisions par des multiplications. On n'effectue ainsi que n divisions. Cette économie est d'autant plus importante que les matrices sont petites.

Cette transformation change les résultats du fait que multiplier par l'inverse nécessite un arrondi intermédiaire. Ainsi, x/y est arrondi une seule fois et a donc une erreur < 0.5 ulp, alors que $x \cdot (1/y)$ est arrondi deux fois et, par conséquent, a une erreur plus grande pouvant atteindre 1 ulp. La différence est, en pratique, négligeable tant que la compatibilité IEEE 754 n'est pas requise.

Afin d'accélérer d'avantages ces opérations, les optimisations vues dans la sous-section 2.2.2 sont appliquées.

4.3 Analyse des résultats

4.3.1 Protocole

Afin de mesurer l'impact de chaque transformation, nous avons procédé à des mesures exhaustives. La performance du code a été mesurée sur six machines dont les spécifications sont données dans la tableau 4.2. Cette table présente les caractéristiques techniques des machines telles que le nombre de cœurs, la fréquence, la taille des caches et leur bande passante. Cette table montre également le parallèlisme intrinsèque des machines par cœur (π) et pour toute la machine (Π). Π représente le nombre d'opérations arithmétiques que la machine est capable d'effectuer à chaque cycle.

Sur x86, le code a été compilé avec le compilateur icc v18.0 et les options suivantes : `-std=c99 -O3 -vec -ansi-alias`. Le temps a été mesuré en cycles normalisés avec l'instruction `RDTSC`.

Sur les autres architectures (ARM et Power), gcc 7.2 a été utilisé avec les options suivantes : `-std=c99 -O3 -ffast-math -fstrict-aliasing`. Le temps a été mesuré avec `clock_gettime(CLOCK_MONOTONIC, ... )`.

Trois fonctions ont été mesurées :

***factorize* :** Factorisation uniquement : $A \to L \cdot L^{\intercal}$.

***substitute* :** Résolution du système (remontée + descente) : $L \cdot L^{\intercal} \cdot X = R$.

***solve* :** Résolution complète du système (`factorize` + `substitute`) : $A \cdot X = R$.

Table 4.2 – Machines testées

CPU		HSW	i9	SKX	KNL	EPYC	A72	Power8
Référence		E5-2683 v3	i9-7900X	Platinum 8168	Phi 7220	7351P	Cortex A72	Turismo
Constructeur		Intel	Intel	Intel	Intel	AMD	ARM	IBM
Fréquence (GHz)		2.0	3.3	2.7	1.3	2.4	2.4	3.0
Cœurs/*threads*		2× 14/28	10/20	2× 24/48	64/256	16/32	2× 32/32	4/32
ISA		AVX2	AVX512	AVX512	AVX512	AVX2	Neon	VSX
Largeur SIMD		256	512	512	512	256	128	128
FMA/cycle/cœur		2	2	2	2	1	1	2
π F32 (op/cycle/cœur)		32	64	64	64	16	8	16
Π F32 (op/cycle)		896	640	3072	4096	256	512	64
L1 (Ko, par cœur)		32	32	32	32	32	32	64
L2 (Ko, par cœur)		256	1024	1024	512	512	256	512
L3 (Mo, par CPU)		35	13	32	16384*	64	32	8
Bande passante aggrégée (Go/s)	L1	5360	4390	14640	6140	1470	2520	324
	L2	1364	1130	3640	1200	1340	718	280
	L3	576	240	760	384*	1060	376	217
	RAM	108	65	208	82	138	96	20
ratio Gflop/Go	L1	2	2	2	2	2	2	2
	L2	7.9	7.8	8.0	10	2.2	7.0	2.3
	L3	19	37	38	32*	2.8	13	3.0
	RAM	100	135	140	150	21	52	32

$*$: mémoire secondaire utilisant la HBM (*High Bandwidth Memory*)
π : parallèlisme intrinsèque par cœur
Π : parallèlisme intrinsèque total

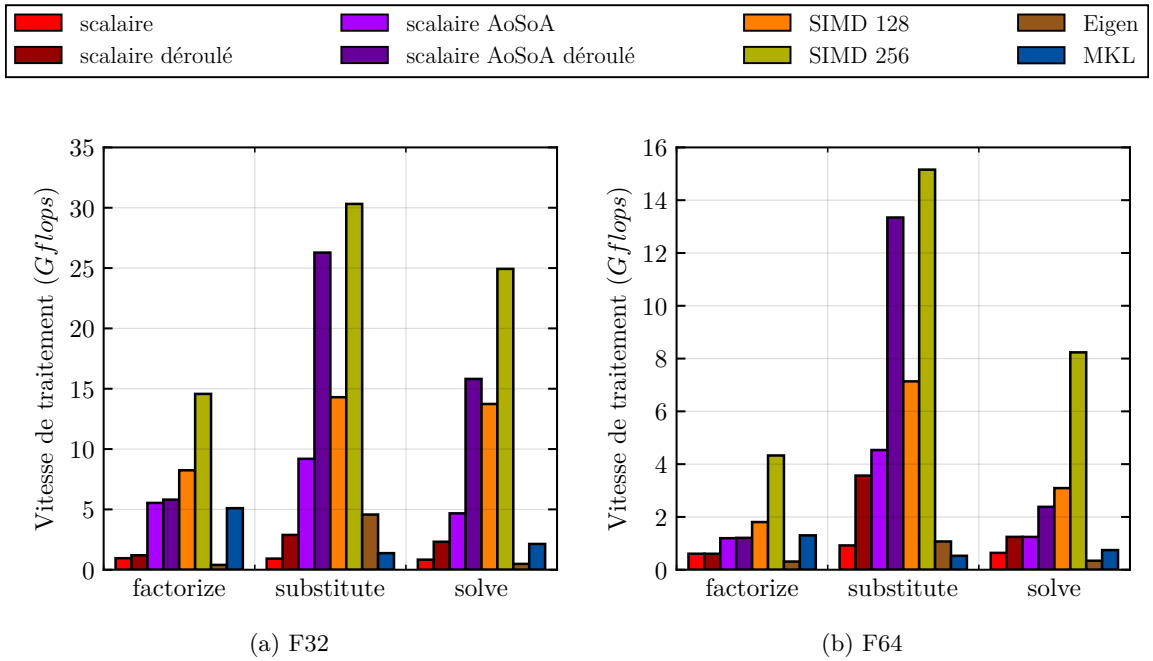
Comme le temps d'exécution de ces fonctions ne dépend pas des données, l'analyse sur la mesure du temps effectuée dans la section 3.2 est donc applicable. Ainsi, le code est exécuté plusieurs fois, et le meilleur temps est sélectionné. On s'assure ainsi que le cache est chaud.

Les courbes utilisent les conventions suivantes :

— `scalaire` : correspond au code scalaire (peut être vectorisé par le compilateur),
— `SIMD` : correspond au code écrit avec les *intrinsics* SIMD,
— `déroulé` : les boucles internes sont entièrement déroulées et scalarisées,
— `classique` : l'inverse n'est pas stocké (version de base),
— `rapide` : la racine carrée inverse rapide est utilisée,
— `approximé` : "`rapide`" sans itération de Newton-Raphson,
— `xk` : ordre de déroulage-entrelacement (*unroll&jam*) de la boucle externe (`x1` signifie pas de déroulage).

Les bibliothèques Intel MKL et Eigen ont également été testées.

Les explications sont centrées sur la fonction `solve` en simple précision sur la machine HSW. Lorsque rien n'est dit à propos des autres fonctions, de la double précision ou des autres architectures, cela indique un comportement similaire.



(a) F32 (b) F64

Figure 4.1 – Vitesse de traitement de Cholesky 3×3 (HSW)

4.3.2 Performance des différentes fonctions

La figure 4.1 montre la vitesse de traitement des fonctions *factorize*, *substitute* et *solve*. La vitesse de traitement reportée est la vitesse de traitement maximale observée.

La vitesse de traitement en simple précision est deux fois plus haute que la vitesse de traitement en double précision pour toutes les versions *AoSoA* de la fonction *substitute*. En revanche, les fonctions *factorize* et *substitute* ont une accélération plus forte : les instructions division et racine carrée sont plus lentes en double précision et la racine carrée inverse rapide nécessite plus de calculs pour affiner la précision.

La vitesse de traitement des versions AVX (`SIMD 256`) est le double des versions SSE (`SIMD 128`).

La performance de la version scalaire *AoSoA* déroulée est proche de la performance de la meilleure version SIMD pour la fonction *substitute* ($\simeq 15\%$). Cependant, la performance pour les autres fonctions est plus basse : l'optimisation de la racine carrée inverse que l'on a utilisé pour la version SIMD n'est pas aussi bien implémentée par le compilateur. Cela est d'autant plus visible en double précision (figure 4.1b) où les instructions division et racine carrée sont plus lentes.

Les bibliothèques d'algèbre linéaire (MKL et Eigen) sont lentes pour toutes les fonc-

tions. Celles-ci sont optimisées pour des matrices beaucoup plus grandes, mais de telles optimisations ne donnent aucun gain pour des matrices aussi petites. Eigen utilise des *templates*, ce qui lui permet d'être un peu meilleur pour *substitute*. Le compilateur sera, en effet, capable d'*inline* les fonctions appelées et d'optimiser aggressivement le tout. La MKL, en revanche, est plus rapide pour *factorize* et *solve* grâce à son interface *compact*. Cette dernière correspond en pratique à un agencement mémoire *AoSoA* qui permet la vectorisation malgré la petite taille des matrices.

4.3.3 Influence de la taille des données

La figure 4.2 présente la vitesse de traitement en fonction de la taille du *batch* pour des matrices 3×3.

En regardant la figure 4.2a, on observe d'abord une augmentation de la vitesse de traitement, puis 3 chutes. On peut ainsi découper la courbe en sept zones distinctes :

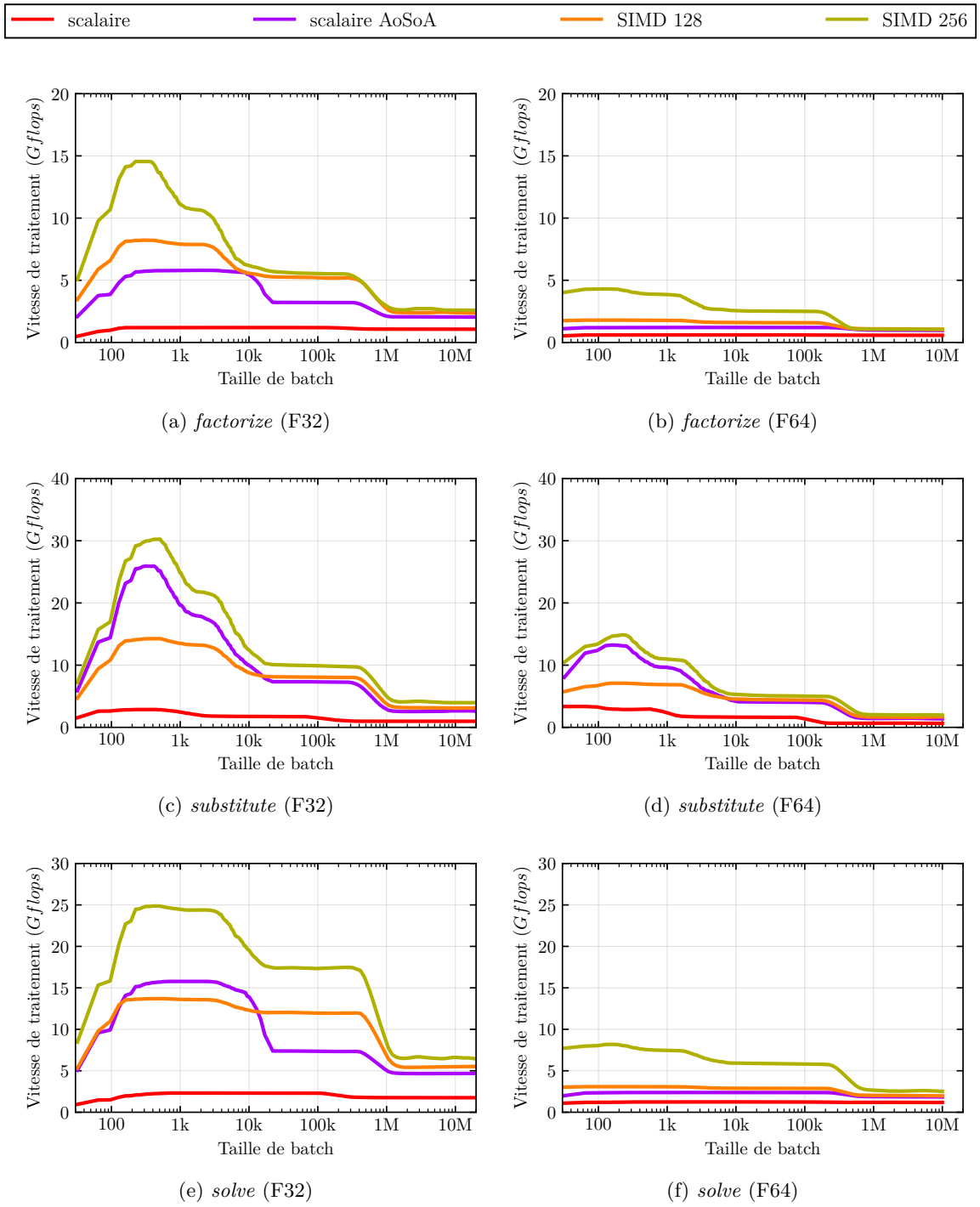
- $[1, 300]$: remplissage des registres SIMD et amortissement des coûts constant : la vitesse de traitement augmente
- $[300, 1\text{k}]$: Sortie du cache L1 : le cache L1 ayant une capacité de 32 Ko, il n'y a de la place que pour 682 systèmes
- $[1\text{k}, 3\text{k}]$: La vitesse de traitement est limitée par le débit du cache L2
- $[3\text{k}, 20\text{k}]$: Sortie du cache L2 : le cache L2 ayant une capacité de 256 Ko, il n'y a de la place que pour 5461 systèmes
- $[20\text{k}, 300\text{k}]$: La vitesse de traitement est limitée par le débit du cache L3
- $[300\text{k}, 1.5\text{M}]$: Sortie du cache L3 : le cache L3 ayant une capacité de 35 Mo, il n'y a de la place que pour 764 586 systèmes
- $[1.5\text{M}, \infty)$: La vitesse de traitement est limitée par le débit de la mémoire externe

On remarque l'absence d'une zone où la vitesse de traitement est limitée par le débit du cache L1 : les coûts constants sont trop important pour être complètement amortis avant la sortie de cache L1.

Certaines versions comme la version scalaire *AoSoA* ne présentent pas de sortie de cache L1 : la vitesse de traitement n'est donc pas limitée par le débit du cache L2.

En double précision (figure 4.2b), la situation est similaire. On peut toutefois noter l'absence de "pic" : les coûts constants sont amortis plus vite de par l'exécution plus lente de la fonction. On remarque aussi que les sorties de caches surviennent pour des *batchs* deux fois plus petits : un `double` prenant deux fois plus de place en mémoire, le cache sera rempli deux fois plus vite.

Les fonctions *factorize* et *substitute* sont similaires.



(a) *factorize* (F32)

(b) *factorize* (F64)

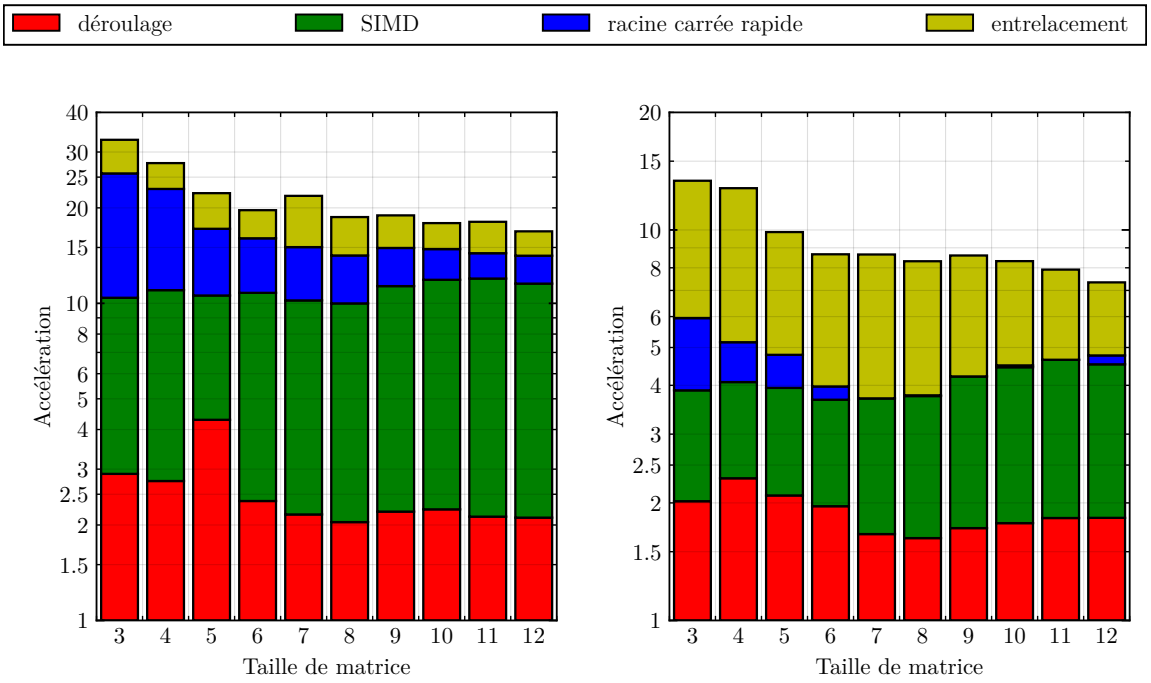
(c) *substitute* (F32)

(d) *substitute* (F64)

(e) *solve* (F32)

(f) *solve* (F64)

Figure 4.2 – Performance de Cholesky en fonction de la taille du *batch* (HSW)



(a) *solve* (F32) (b) *solve* (F64)

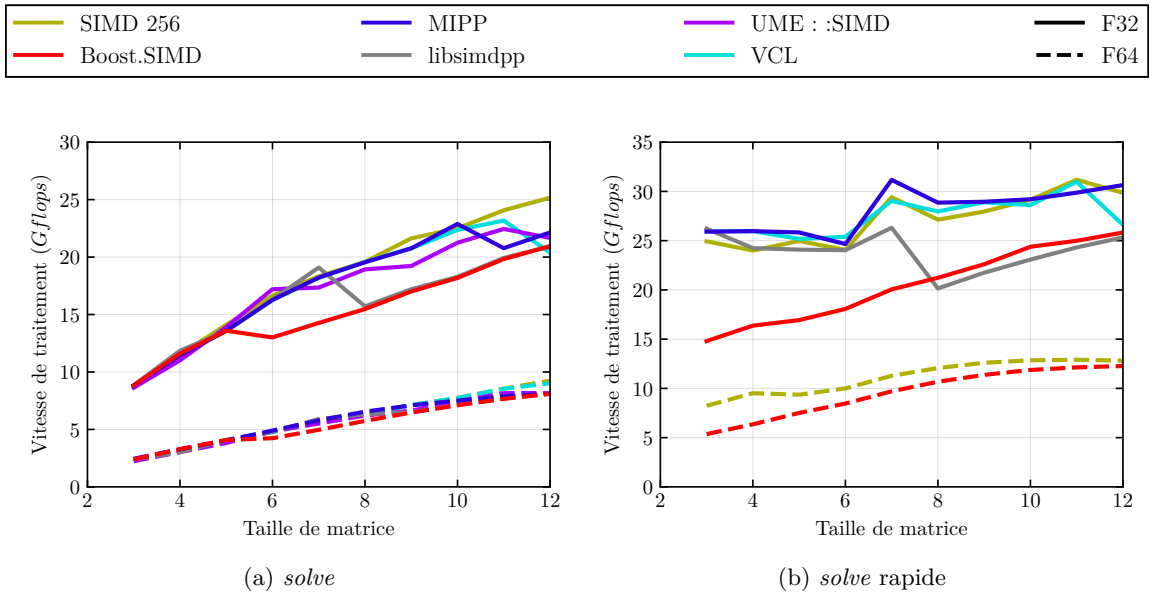
Figure 4.3 – Accélération cumulée des transformations appliquées à Cholesky (HSW)

4.3.4 Influence des optimisations

La figure 4.3 montre l'accélération cumulée de chaque transformation dans l'ordre suivant : déroulage complet, *AoSoA* + SIMD, racine carrée rapide, déroulage–entrelacement. L'accélération d'une transformation dépend des transformations déjà appliquées : un ordre différent répartira les accélérations différemment.

Le déroulage complet permet d'accélérer le code d'un facteur de $\times 2$ à $\times 3$. Ce gain décroit lorsque la taille des matrices augmente : la pression de registres augmente. En regardant l'assembleur, on peut voir que le compilateur génère du *spill code* pour les grandes matrices.

Le SIMD donne une accélération sous-linéaire : de $\times 3$ à $\times 6$ en simple précision. Dans les faits, la fonction *solve* ne peut pas bénéficier complètement du SIMD sans recourir à la racine carrée inverse rapide. Une analyse plus poussée de ce graphique permet de voir que l'accélération SIMD + RSQRT rapide est entre $\times 7$ et $\times 8$. L'impact de la racine carrée rapide diminue lorsque la taille des matrices augmente : leur proportion devient négligeable face aux autres opérations arithmétiques. Le SIMD seul obtient donc une accélération plus grande.



(a) *solve* (b) *solve* rapide

Figure 4.4 – Comparaison de la vitesse de traitement de Cholesky entre *wrappers* SIMD (HSW)

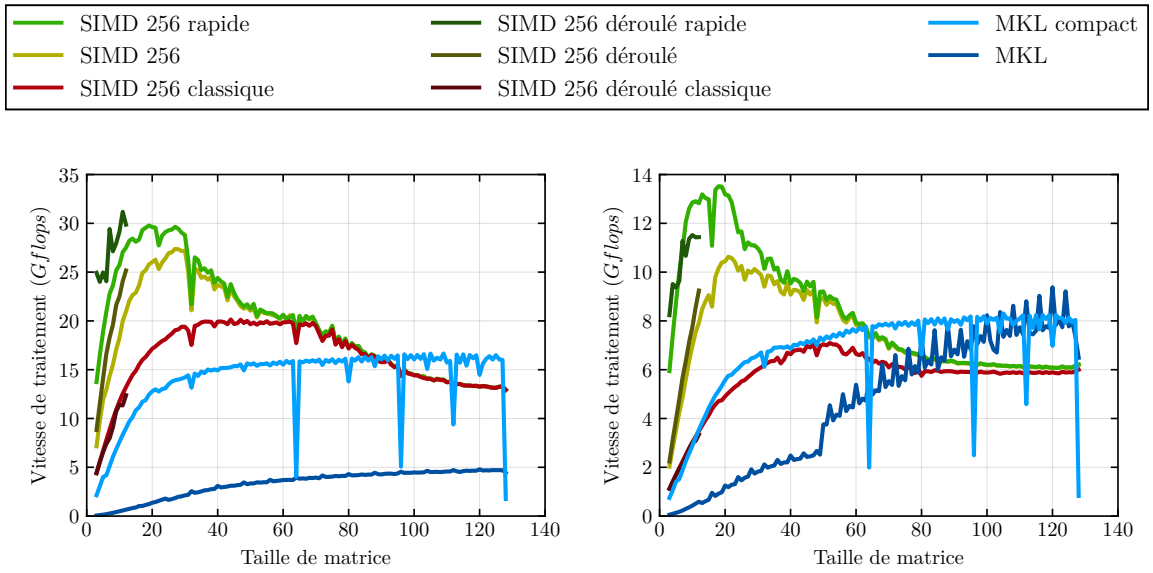
En simple précision, dérouler–entrelacer n'a qu'un impact faible ($\simeq 20\%$). En revanche, le gain est important en double précision ($\times 2$) : le raffinement de l'approximation de la racine carrée forme une longue chaîne de dépendance que l'entrelacement permet d'atténuer. De plus, la transformation racine carrée rapide + dérouler–entrelacer a un impact bien supérieur en double précision dû à la lenteur des instructions division et racine carrée pour cette précision.

4.3.5 Comparaison avec les *wrappers* SIMD

La figure 4.4 montre les différences de vitesse de traitement des différentes bibliothèques SIMD (Boost.SIMD, MIPP, libsimdpp, UME::SIMD et Vcl) et de la version utilisant les *intrinsics*. Les transformations qui ne sont pas spécifiques au SIMD sont appliquées à toutes les versions.

Toutes les bibliothèques ont des performances similaires pour de petites matrices. À partir d'un certain point, la vitesse de traitement chute. Ce point est différent pour chaque bibliothèque. Il s'agit d'un bug du compilateur qui arrête d'*inline* les fonctions de la bibliothèque lorsque la fonction appelante est trop longue, suite aux déroulages.

La version "rapide" en simple précision présente un comportement similaire. Cependant, UME::SIMD n'implémente pas l'approximation de la racine carrée inverse, bien que cette



(a) *solve* (F32) (b) *solve* (F64)

Figure 4.5 – Comparaison de la vitesse de traitement de Cholesky avec la MKL (HSW)

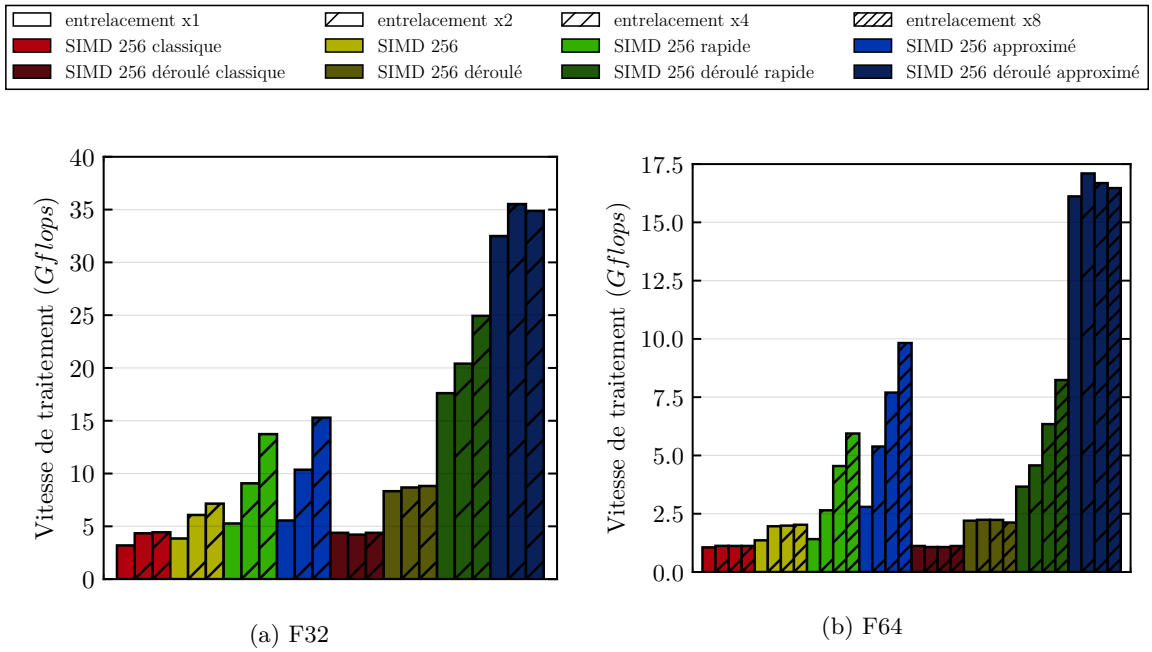
fonction soit présente dans la documentation. De plus, seul Boost.SIMD supporte la racine carrée inverse en double précision. Dans ce cas, le code *intrinsics* est plus rapide.

4.3.6 Comparaison avec la MKL

La version 2018 de la MKL supporte désormais l'agencement mémoire *AoSoA* au travers des fonctions "compact". Avant cela, la MKL ne supportait que l'agencement mémoire *AoS* qui est beaucoup moins adapté à de si petites matrices, et ne pouvait traiter des *batchs* de matrices. La comparaison de la vitesse de traitement entre notre implémentation et la MKL est montrée dans la figure 4.5.

La version "compact" améliore la vitesse de traitement pour les petites matrices, relativement à l'ancienne version de la MKL. Elle reste cependant plus lente que notre version pour des matrices plus petites que 90×90 en simple précision. Pour la double précision, la MKL devient plus rapide à partir de matrices 60×60.

La MKL n'enregistre pas l'inverse et doit donc calculer des divisions lors de la factorisation ainsi que de la substitution. Celà est comparable à la version"classique". La MKL utilise également un algorithme récursif pour la descente et la remontée (*substitute*) qui n'est pas efficace pour des petites matrices.



(a) F32

(b) F64

Figure 4.6 – Influence du déroulage sur la vitesse de traitement de Cholesky 3×3 (HSW)

4.3.7 Influence du déroulage

On peut voir l'influence du déroulage sur la vitesse de traitement sur la figure 4.6. Sans aucun déroulage, toutes les versions atteignent une vitesse de traitement similaire : la vitesse semble limitée par la latence des instructions due à une forte dépendance de données. Le déroulage complet permet alors d'aider le réordonnanceur et ainsi de réduire les dépendances locales.

La vitesse des versions normale et "classique" est limitée par la cadence des instructions division et racine carrée : on observe un plateau pour ces versions malgré les déroulages. Il est alors impossible de rendre ces versions plus rapide sans diminuer le nombre de ces instructions. Il est alors indispensable de recourir au calcul rapide de la racine carrée inverse pour s'affranchir de cette limite. La version "classique" est encore plus limitée de par la présence de plus de divisions. En double précision, cette limite est encore plus basse du fait d'instructions division et racine carrée plus lentes. Pour les versions "rapide", les deux techniques de déroulage sont efficaces. Dérouler–entrelacer permet de réduire les *pipeline stalls* entre les instructions dépendantes. Cette transformation permet ainsi d'atteindre une accélération d'un facteur ×3 sur le code non-déroulé, et d'un facteur ×1.5 lorsque les boucles sont totalement déroulées. Du fait d'une pression de registres plus importante, dérouler–entrelacer est moins efficace sur du code déjà déroulé.

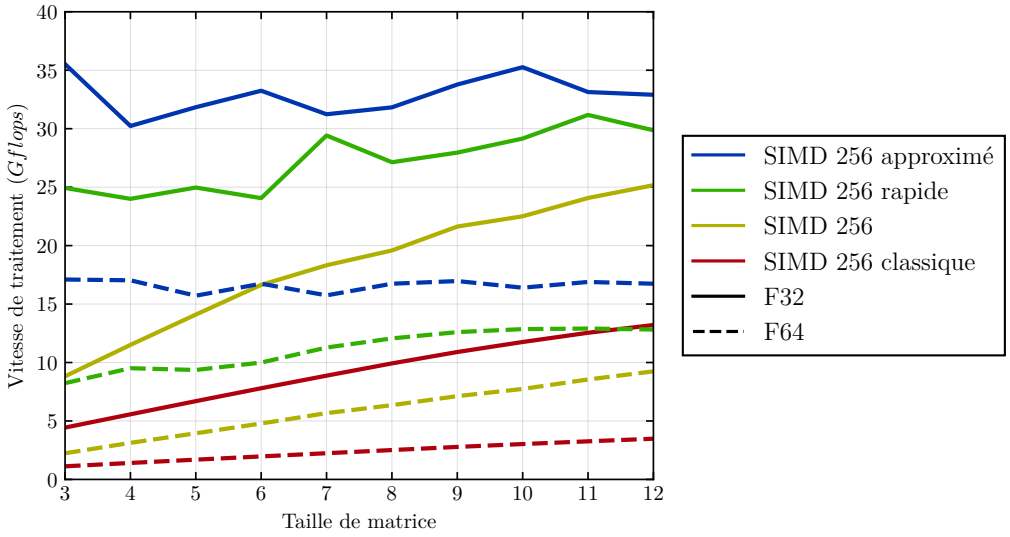


Figure 4.7 – Influence de la taille des matrices sur la vitesse de traitement de Cholesky (HSW)

La version “approximée” déroulée est bien plus rapide, surtout en double précision. Il est en effet possible d’économiser beaucoup d’instructions s’il n’y a pas besoin de raffiner l’approximation de la racine carrée inverse. Comme ce raffinement nécessite plus d’instructions en double précision qu’en simple, il est normal d’observer un gain plus important en double précision.

4.3.8 Influence de la taille des matrices

Lorsque l’on regarde la vitesse de traitement en fonction de la taille des matrices (figure 4.7), on peut voir que la vitesse augmente avec la taille des matrices, sauf pour la version approximée. L’intensité arithmétique augmente pour des matrices plus grandes.

L’augmentation de cette vitesse est plus grandes pour les versions normale et “classique”, la proportion de racines carrées et divisions diminue.

4.3.9 Autres architectures

Les vitesses de traitement pour toutes les machines testées sont visibles dans la figure 4.8 et la figure 4.9.

Sur la machine A72, gcc n’est pas capable de vectoriser notre code scalaire. L’écriture de code SIMD est alors indispensable.

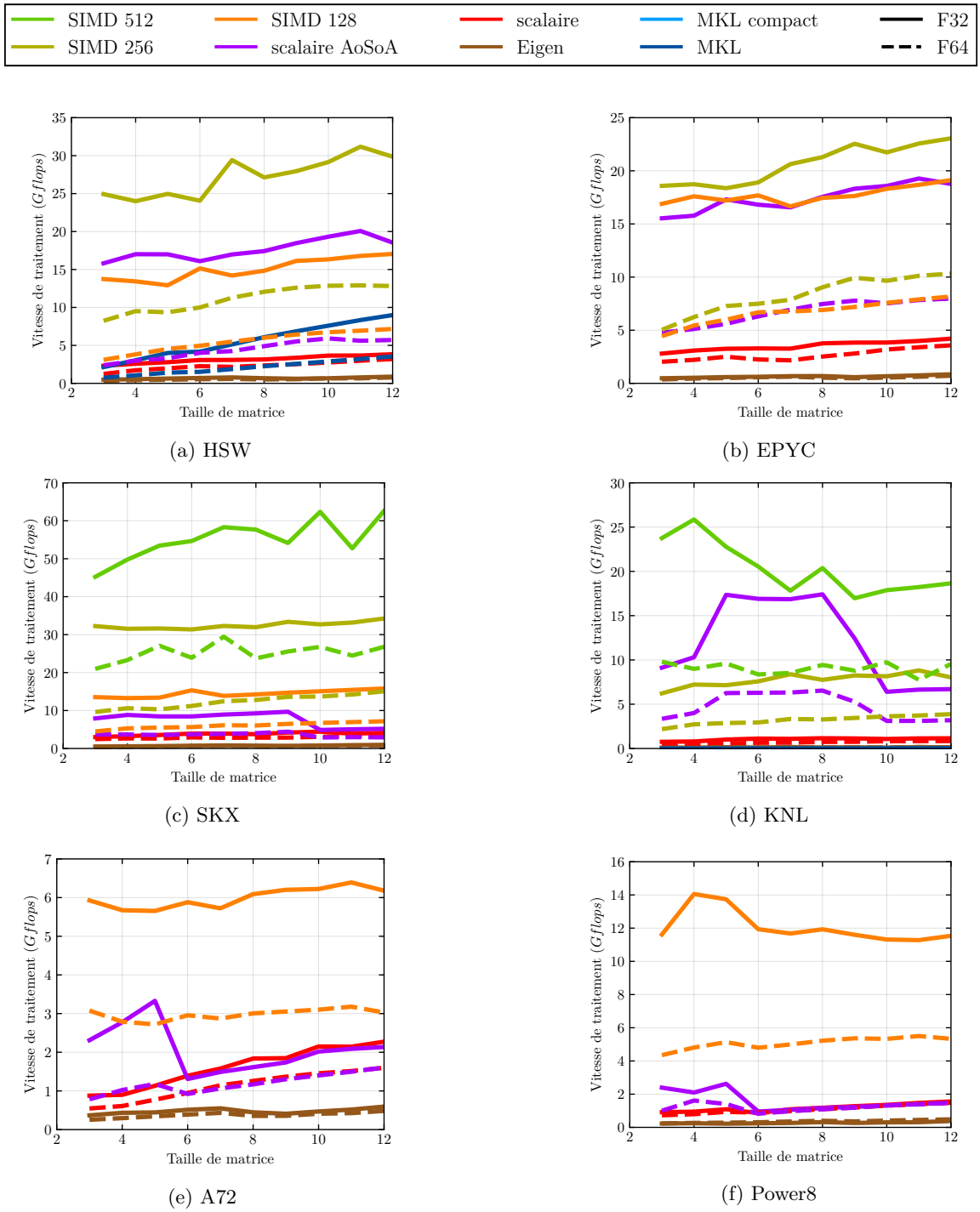


Figure 4.8 – Vitesse de traitement de Cholesky (1 cœur)

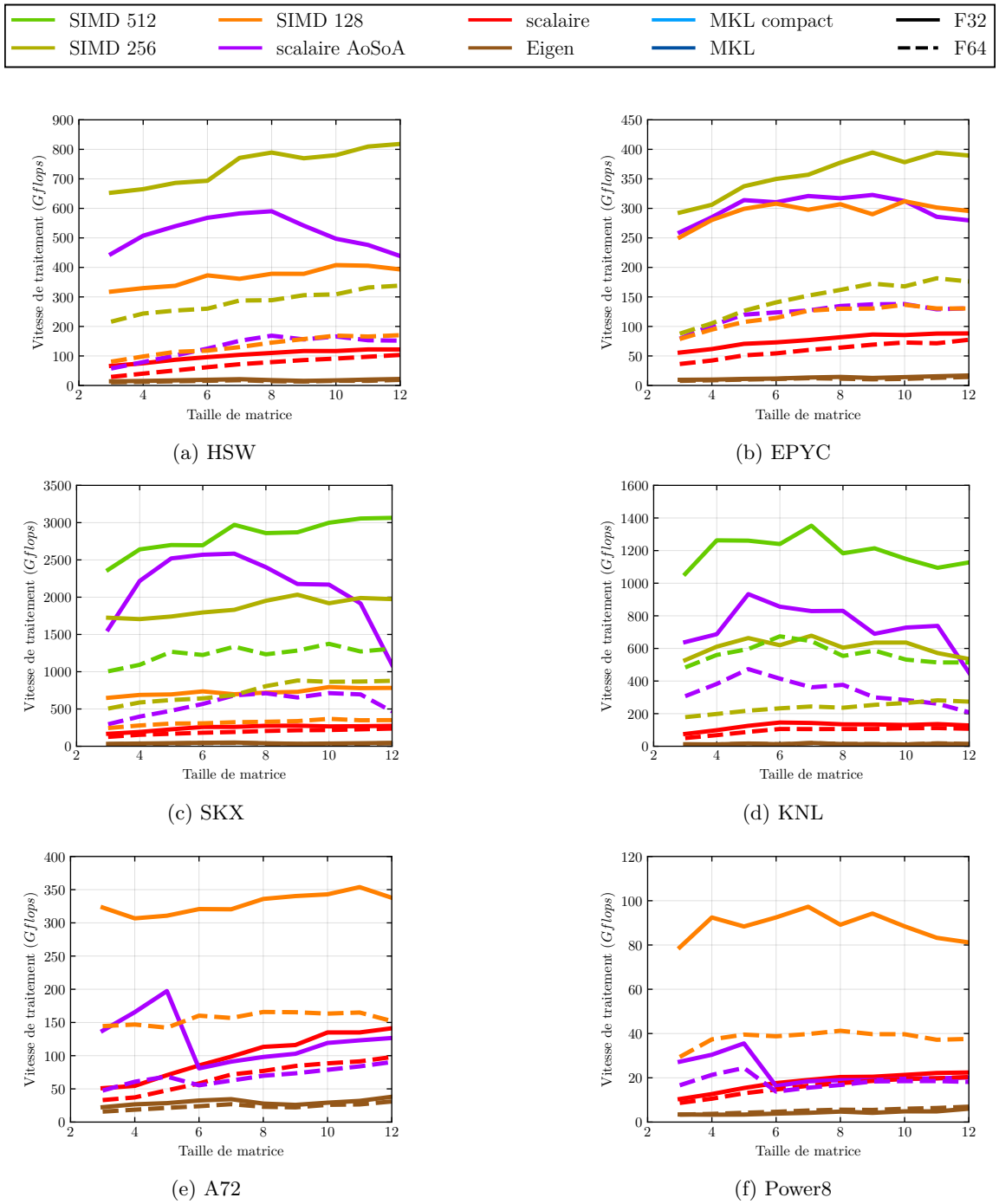
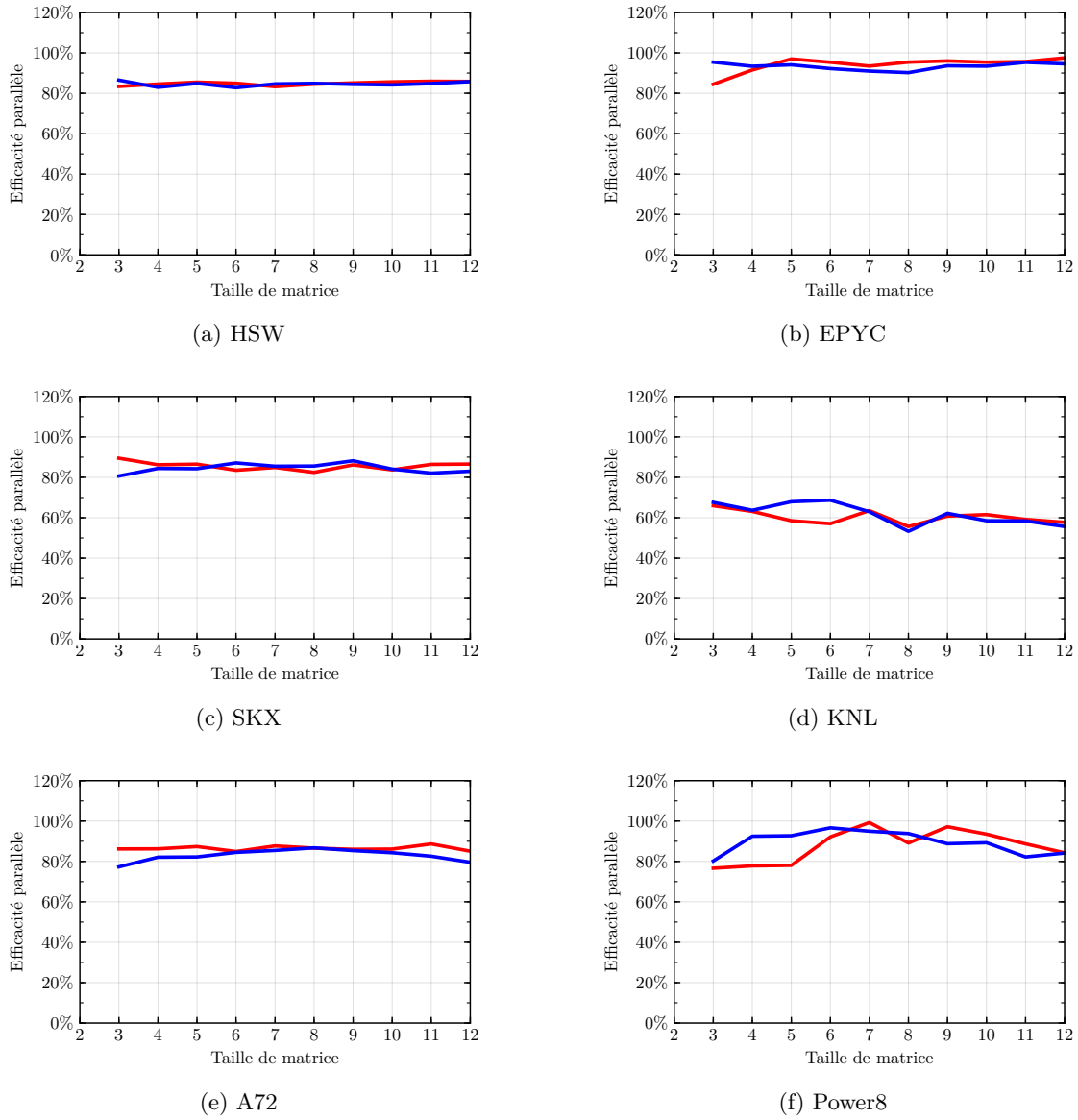


Figure 4.9 – Vitesse de traitement de Cholesky (multi-cœur)

4.3.10 Passage à l'échelle : OpenMP

Le passage à l'échelle est bon sur toutes les machines ($\geqslant$ 80%), excepté sur KNL (figure 4.10). Il est en effet plus difficile de nourrir tous les cœurs sur cette architecture *many-core*.



(a) HSW (b) EPYC (c) SKX (d) KNL (e) A72 (f) Power8

Figure 4.10 – Efficacité parallèle de Cholesky (`float` – `double`)

4.4 Synthèse

Dans cette partie, nous avons implémenté des transformations à la résolution de systèmes par la factorisation de Cholesky pour des petites matrices (de 3×3 à 12×12).

La petite taille des matrices impose de traiter des *batchs* de systèmes. L'agencement mémoire *AoSoA* permet alors une utilisation efficace des unités SIMD. Le déroulage total accompagné de la scalarisation permet d'augmenter fortement l'Intensité Arithmétique. La dernière transformation est l'utilisation de la racine carrée inverse rapide qui permet de s'affranchir des instructions lentes de division et de racine carrée. Sans cette transformations, la vitesse de traitement de Cholesky plafonne.

Le tableau 4.3 montre l'accélération de la meilleure version de Cholesky par rapport à la version *AoS* naïve pour chaque machine testée. Ainsi, sur HSW, on obtient une accélération de ×33 pour des matrices 3×3 simple précision et ×13 en double précision. L'accélération sur KNL n'est pas représentative du fait de la comparaison avec le code scalaire, KNL n'étant pas adapté pour l'exécution de code scalaire.

La meilleure implémentation en simple précision sur HSW est ainsi ×10 plus rapide que la MKL en 3×3 et ×3 plus rapide en 12×12. Le passage à l'échelle est bon avec une efficacité multi-cœur > 80%.

Table 4.3 – Accélération de la meilleure version SIMD par rapport à la version *AoS* naïve pour Cholesky jusqu'à 12×12

Machine	mono-cœur		multi-cœur	
	F32	F64	F32	F64
HSW	×17 – ×33	×7.3 – ×13	×16 – ×29	×7.0 – ×12
i9	×31 – ×54	×14 – ×33	×24 – ×43	×10 – ×23
SKX	×33 – ×57	×14 – ×36	×26 – ×49	×12 – ×26
KNL	×105 – ×425	×46 – ×170	×50 – ×150	×23 – ×68
EPYC	×13 – ×29	×6.3 – ×8.9	×11 – ×21	×5.7 – ×6.9
A72	×6.7 – ×15	×3.2 – ×10	×6.2 – ×16	×2.7 – ×8.4
Power8	×14 – ×37	×7.1 – ×15	×6.3 – ×17	×3.1 – ×7.0

Chapitre 5

Filtre de Kalman

Le filtre de Kalman [88, 89] est un filtre beaucoup utilisé en physique des hautes énergies [90], *tracking* [91], vision par ordinateur [92, 93], traitement du signal [94], positionnement par satellite [95], robotique [96] ou bien encore en Économie [97].

L'accélération du filtre de Kalman n'est pas un problème récent [98]. Plusieurs équipes ont déjà accéléré la vitesse de ce filtre sur des architectures SIMD [99] ou GPU [100].

En physique des particules, le filtre de Kalman est utilisé pour estimer les paramètres de la trajectoire d'une particule et calculer la vraisemblance de la trajectoire. L'état à estimer est de faible dimension (à LHCb : 5), mais le filtre est appliqué à un nombre important de particules. Ce problème est *embarrassingly parallel*.

Au sein de LHCb, la reconstruction des évènements (HLT1) est dominée par ce filtre de Kalman (`ForwardFitterAlgParam` dans la figure 5.1). Le filtre à lui seul nécessite 47% de la séquence. C'est donc un algorithme de choix afin d'accélérer la séquence complète.

Dans cette partie, nous verrons d'abord la formulation générale du filtre de Kalman. Les différentes transformations appliquées y seront expliquées et analysées. Puis nous verrons la formulation spécifique à LHCb du filtre de Kalman, et le gain apporté par l'application des transformations vues précédemment.

5.1 Algorithme

Le filtre de Kalman est un algorithme itératif permettant d'estimer l'état d'un système dont l'évolution est bruitée à partir de mesures bruitées ou incomplètes. Il est communément utilisé en physique des hautes énergies et en vision artificielle pour reconstruire la trajectoire d'un objet (*tracking*).

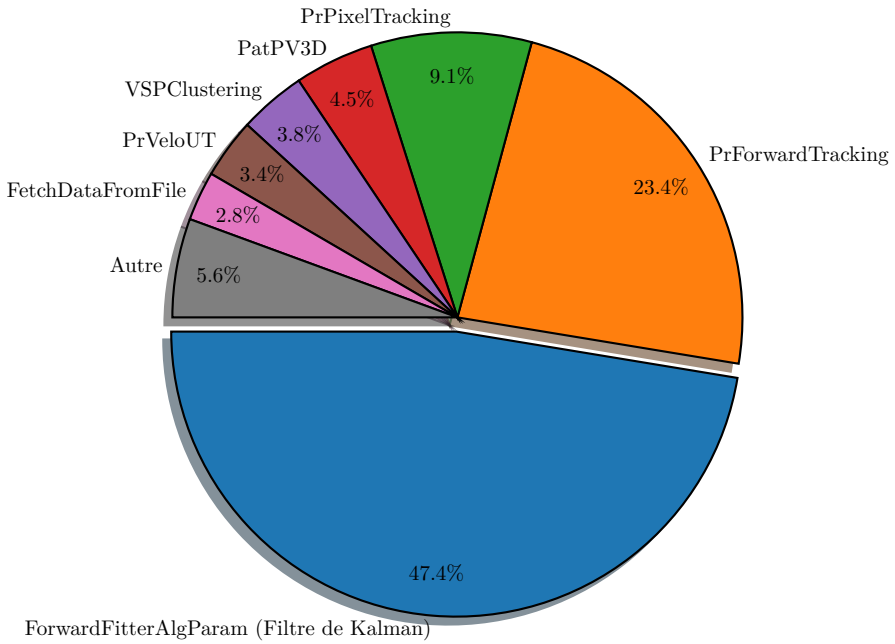


Figure 5.1 – Proportion du temps passé dans les différents algorithmes de la séquence de reconstruction “HLT1” au sein de LHCb

Le filtre de Kalman consiste en plusieurs multiplications matricielles, ainsi que d'une inversion matricielle pouvant être calculée avec la factorisation de Cholesky (algorithme 5.1).

Le filtre de Kalman travaille sur 3 espaces distincts : l'espace des états, l'espace des commandes et l'espace des mesures. Ces espaces peuvent avoir un nombre de dimensions différent. L'état correspond aux différentes grandeurs nécessaires à caractériser le système et prédire son évolution. La commande permet de mieux prédire l'état lorsqu'une action connue est appliquée au système. La mesure correspond à ce que l'on observe du système.

La matrice A est la matrice d'évolution du système : Ax sera l'état suivant l'état x s'il n'y avait ni bruit ni commande. La matrice B correspond à l'influence de la commande sur le système. La matrice H est la matrice de projection de l'espace des états vers l'espace des mesures. Les matrices Q et R sont des matrices de bruit : respectivement le bruit du processus et le bruit de la mesure.

Illustrons ces différents espaces avec un exemple simple : le GPS d'une voiture. L'état considéré sera la position et la vitesse de la voiture : $(x, y, \dot{x}, \dot{y})$. La commande sera l'enfoncement de l'accélérateur : (a). Et la mesure sera la position donnée directement par la géolocalisation satellite : $(\hat{x}, \hat{y})$.

Algorithme 5.1 Filtre de Kalman (v1)

E/S : x, P // état, covariance
entrée : u, z // commande, mesure
entrée : A, B, H, Q, R // Paramètres du filtre

// Prédiction
1 $x' \leftarrow A\,x + B\,u$
2 $P' \leftarrow A\,P\,A^{\intercal} + Q$

// Innovation
3 $\tilde{y} \leftarrow z - H\,x'$
4 $S \leftarrow H\,P'\,H^{\intercal} + R$

// Gain de Kalman
5 $K \leftarrow P'\,H^{\intercal}\,S^{-1}$

// Mise à jour
6 $x \leftarrow x' + K\,\tilde{y}$
7 $P \leftarrow (I - K\,H)\,P'$

$$A = \begin{pmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1+dv & 0 \\ 0 & 0 & 0 & 1+dv \end{pmatrix} \quad B = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \quad H = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad \text{où} \begin{cases} dt & = \text{pas de temps} \\ dv & = \text{decéleration} < 0 \end{cases} \tag{5.1}$$

$$Q = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \sigma_v^2 & 0 \\ 0 & 0 & 0 & \sigma_v^2 \end{pmatrix} \quad R = \begin{pmatrix} \sigma^2 & 0 \\ 0 & \sigma^2 \end{pmatrix} \quad \text{où} \begin{cases} \sigma^2 & = \text{variance sur la mesure de la position} \\ \sigma_v^2 & = \text{variance de l'évolution de la vitesse} \end{cases} \tag{5.2}$$

Une première analyse est faite sur un système dont les trois espaces sont de dimension 4 afin de valider l'implémentation du filtre.

5.2 Transformations

Toutes les transformations vues jusque-là sont également appliquées au filtre de Kalman. On retrouve ainsi le déroulage complet, le déroulage avec entrelacement, l'agencement mémoire *AoSoA*, la racine carrée rapide et le traitement par lot. De nouvelles transformations ont également été testées.

5.2.1 Optimisation des accès aux matrices symétriques

Le filtre de Kalman nécessite de manipuler des matrices symétriques définies positives (P, S, Q et R). De par leur symétrie, il est possible de réduire le nombre d'accès mémoires en accédant qu'à une moitié de la matrice (triangle inférieur ou supérieur). Lorsque le résultat est une matrice symétrique, il est également possible d'économiser des opérations arithmétiques en ne calculant qu'une moitié.

Lorsque les matrices sont en *AoS*, accéder seulement à la moitié de la matrice réduit l'efficacité de la vectorisation, en particulier pour les petites matrices. En effet, l'accès aux éléments proches de la diagonal n'est pas régulier.

Cependant, lorsque les matrices sont en *SoA*, on s'affranchit d'une telle pénalité : chaque accès peut se faire sur un registre SIMD complet. Ainsi, la vectorisation est aussi efficace que pour les matrices carrées, et nécessite moins d'opérations et moins d'accès mémoires.

5.2.2 Transformations algébriques

La première transformation algébrique à considérer est la réutilisation de résultats intermédiaires. Ainsi, la matrice produit $\Gamma = P' H^\intercal$ utilisée dans le calcul de S (ligne 4) peut être réutilisée pour le calcul de K (ligne 5).

Il est également intéressant de garder S sous sa forme factorisée et de propager l'expression de K dans les expressions de x et P.

$$\begin{aligned} x &= x' + P' H^\intercal L^{-1\intercal} L^{-1} \tilde{y} \\ P &= \left(I - P' H^\intercal L^{-1\intercal} L^{-1} H\right) P' = P' - P' H^\intercal L^{-1\intercal} L^{-1} H P' \end{aligned} \tag{5.3}$$

Ces expressions sont longues, mais la symétrie de P' permet une importante factorisation :

$$\begin{aligned} M &= L^{-1} H P' = L^{-1} \Gamma^\intercal \\ x &= x' + M^\intercal L^{-1} \tilde{y} \\ P &= P' - M^\intercal M \end{aligned} \tag{5.4}$$

Algorithme 5.2 Filtre de Kalman algébriquement optimisé (v2)

E/S : x, P // état, covariance
entrée : u, z // commande, mesure
entrée : A, B, H, Q, R // Paramètres du filtre

// Prédiction
1 $x' \leftarrow A\,x + B\,u$
2 $P' \leftarrow A\,P\,A^{\intercal} + Q$

// Innovation
3 $\tilde{y} \leftarrow z - H\,x'$
4 $\Gamma \leftarrow P'\,H^{\intercal}$
5 $S \leftarrow H\,\Gamma + R$

// "Gain de Kalman"
6 $L \leftarrow \text{cholesky}(S)$
7 $M \leftarrow L^{-1}\,\Gamma^{\intercal}$

// Mise à jour
8 $x \leftarrow x' + M^{\intercal}\,L^{-1}\tilde{y}$
9 $P \leftarrow P' - M^{\intercal}\,M$

La nouvelle expression de x fait maintenant intervenir un produit à trois termes matrice–matrice–vecteur : $M^{\intercal}\,L^{-1}\,\tilde{y}$. Ce dernier est un cas typique du problème de multiplication matricielles enchaînées [101,102] : il est possible de calculer soit $\left(M^{\intercal}\,L^{-1}\right)\tilde{y}$, soit $M^{\intercal}\left(L^{-1}\,\tilde{y}\right)$.

Dans le cas où les espaces des états et des mesures ont le même nombre de dimensions (M est une matrice carrée $n{\times}n$), la résolution est simple :
— $\left(M^{\intercal}\,L^{-1}\right)\tilde{y}$ a une complexité de $\mathcal{O}\left(n^3\right)$
— $M^{\intercal}\left(L^{-1}\,\tilde{y}\right)$ a une complexité de $\mathcal{O}\left(n^2\right)$
Il faut donc privilégier les produits matrices–vecteurs et évaluer cette expression de la droite vers la gauche.

Toutes ces tranformations permettent de réduire le nombre d'opérations arithmétiques. L'algorithme 5.2 présente ces transformations.

La dernière transformation permettant de réduire le nombre d'opérations arithmétiques est de ne pas inverser L. Il faut alors résoudre un système triangulaire lorsque l'on "multiplie" L^{-1} : résoudre $L\,X = \Gamma^{\intercal}$ permet de calculer $L^{-1}\Gamma^{\intercal}$.

L'algorithme de remontée permettant de résoudre un système triangulaire nécessite autant d'opérations arithmétiques que la mutliplication avec une matrice triangulaire. On économise ainsi le calcul de l'inverse L^{-1} sans ajouter d'opérations par ailleurs. Toutefois, la remontée introduit une longue chaîne de dépendance lors des calculs et possède donc moins de parallèlisme intrinsèque.

Algorithme 5.3 Algorithme de Kahn pour effectuer un tri topologique

```
   entrée : G  // Graphe de dépendance
   entrée : S  // Liste des sommets de G sans dépendances
   sortie : L  // Liste des sommets de G respectant les dépendances
1  L ← {}
2  tant que S non-vide faire
3      retire un sommet s_o de S
4      ajoute s_o à la fin de L
5      pour a parmis arrêtes(G) depuis s_o vers s_d faire
6          retire l'arrête a de G
7          si s_d n'a plus de dépendance alors
8              ajoute s_d à S
9  si arrêtes(G) non-vide alors
10     Graphe cyclique
```

5.2.3 Heuristique de réordonancement

Contrairement à Cholesky, l'implémentation du filtre de Kalman nécessite beaucoup plus d'opérations arithmétiques. Dérouler les boucles augmentera ainsi plus vite la pression de registres. Ordonnancer correctement les instructions permet de diminuer cette pression de registres, et limiter ainsi le *spill code*. Cette tâche est effectuée par le compilateur. Ce problème étant NP-difficile [103], le compilateur a recours à des heuristiques pour allouer les registres.

Il est toutefois possible d'ordonnancer le code source afin d'aider le compilateur à allouer les registres. On peut ainsi utiliser des heuristiques différentes, plus adapatées au problème. Il n'est pas question ici d'allouer les registres à la place du compilateur, mais de rassembler des lignes de code agissant sur les mêmes valeurs afin de diminuer le nombre de valeurs actives en même temps. Ce réordonnancement doit respecter le graphe de dépendance du code.

Un ordonnancement repesctant les dépendances est appelé un tri topologique. L'algorithme de Kahn [104] permet de réaliser un tel tri (algorithme 5.3). Le choix du sommet (ligne 3) à retirer n'influence aucunement la validité du tri topologique. On peut ainsi encoder notre heuristique dans ce choix en attribuant une priorité à chaque sommet.

Chaque nœud du graphe représente une ligne de code. Pour simplifier l'heuristique, chaque ligne de code correspond à une instruction. Un nœud possède deux états : exécuté lorsque le nœud a été ajouté à la liste triée, et non-exécuté dans le cas contraire. Lorsqu'un nœud est "exécuté", une date de fin lui est attribué correspondant au moment où la donnée

sera prête : $temps + latence$. Le temps est également incrémenté. Pour chaque nœud exécuté, on peut alors définir la date à laquelle toutes ses dépendances seront prêtes et comparer cette date avec le temps courant.

Si le temps courant est inférieur à la date, l'exécution d'un tel nœud sera retardée par le processeur afin de respecter la latence des dépendances. Si le temps est supérieur à la date, alors toutes les dépendances sont prêtes, et ce depuis plusieurs cycles. Le cas optimal serait le suivant : chaque nœud est "exécuté" exactement dès que ses dépendances sont prêtes, sans attente additionnelle. Afin de s'approcher de ce cas optimal, les nœuds dont la différence de temps est proche de 0 seront préférés.

Les accès mémoires sont traités différement : une lecture sera toujours "exécutée" juste avant la première ligne dépendant de cette lecture. Inversement, une écriture sera toujours "exécutée" dès que toutes ses dépendances sont "exécutées". Ceci permet d'entrelacer les accèes mémoires avec du calcul et ainsi utiliser au mieux toutes les unités du processeur.

5.3 Analyse des résultats

5.3.1 Protocole

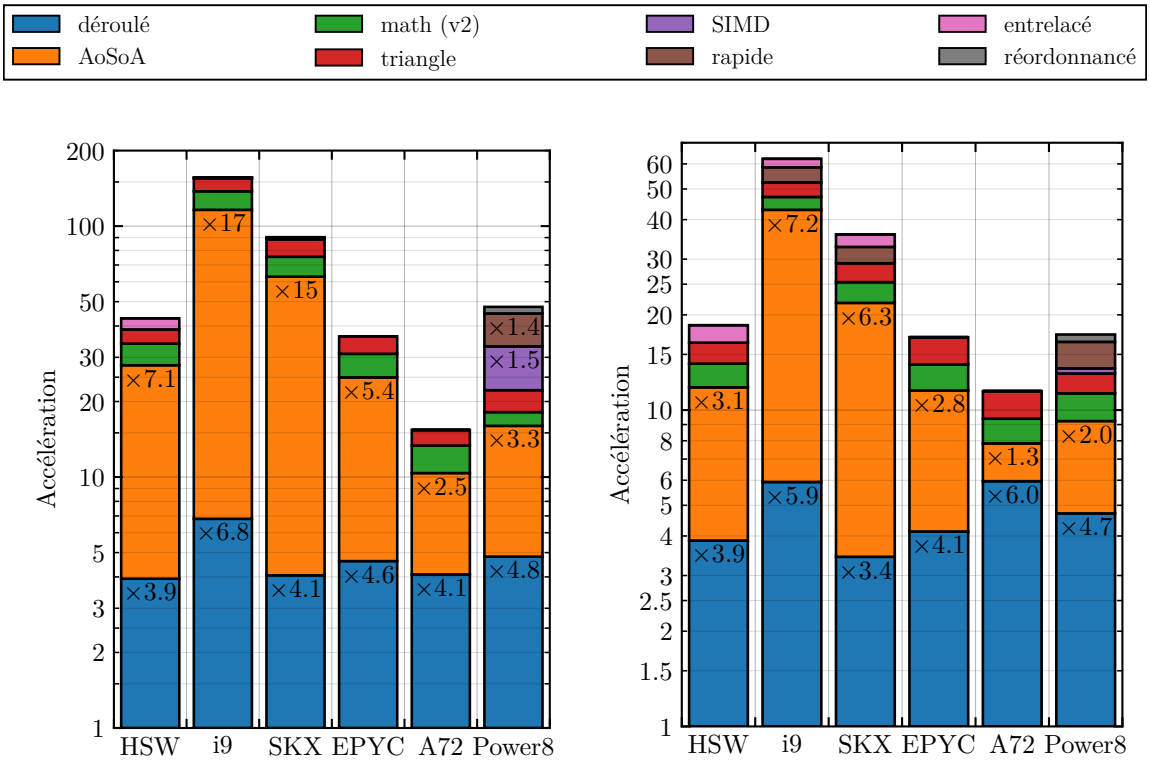
Le protocole utilisé ici est principalement le même que pour la factorisation de Cholesky. Le filtre de Kalman étudié correspond à un système à 4 dimensions $(x, y, \dot{x}, \dot{y})$. L'espace de control et l'espace des mesures sont aussi de dimension 4.

Plusieurs systèmes sont filtrés en parallèle. Le temps est mesuré par itération. Les courbes utilisent les mêmes conventions que pour la factorisation de Cholesky, en addition des suivantes :

- `v1` : La version classique de l'algorithme (algorithme 5.1)
- `v2` : La version optimisée de l'algorithme (algorithme 5.2)
- `triangle` : Les accès aux matrices symétriques ne se font que sur une moitié
- `réordonnancé` : L'heuristique de réordonnancement est appliquée
- `référence` : correspond à la version classique (v1) scalaire *AoS* accédant aux deux moitiés des matrices symétriques

5.3.2 Influence des optimisations

La figure 5.2 montre l'accélération cumulée de chaque transformations appliquées dans l'ordre suivant : déroulage complet, *AoSoA*, version optimisée (v2), l'accès triangulaire aux matrices symétriques, le SIMD, la racine carrée rapide, le déroulage–entrelacement et l'heuristique de réordonnancement.



(a) F32 (b) F64

Figure 5.2 – Impact des transformations sur la vitesse de traitement du filtre de Kalman 4×4

Tout comme avec la factorisation de Cholesky, l'accélération est principalement due au déroulage complet et à l'agencement mémoire *AoSoA* qui permet la vectorisation du code. Les optimizations mathématiques (`v2+triangle`) donnent une accélération totale de ×1.4 en moyenne. Notre heuristique de réordonnancement n'apporte aucun gain visible, sauf sur Power8 où elle permet une accélération de +5%.

Contrairement à la factorisation de Cholesky, la racine carrée rapide et le déroulage–entrelacement ne fournissent presque aucune accélération en simple précision (figure 5.2a). En effet, la proportion de racines carrées et de division est beaucoup plus faible pour le filtre de Kalman. De plus, les opérations sont plus indépendantes entre elles : l'algorithme a plus de parallèlisme intrinsèque. Dérouler–entrelacer n'est pas efficace. Cette dernière transformation reste tout de même intéressante si les boucles n'ont pas été totalement déroulées.

En double précision (figure 5.2b), la longue latence de la racine carrée et la longue

chaîne de dépendance des itérations de Newton-Raphson successives ne sont pas totalement diluées par les autres opérartions arithmétiques. La racine carrée rapide et l'entrelacement restent donc intéressants.

La dernière chose à remarquer est que l'utilisation de SIMD explicite via les *intrinsics* ne permet pas l'amélioration de la vitesse de traitement. La seule exception étant sur Power8 où le compilateur gcc rencontre des difficultés à optimiser le code pour cette architecture.

5.3.3 Performance totale

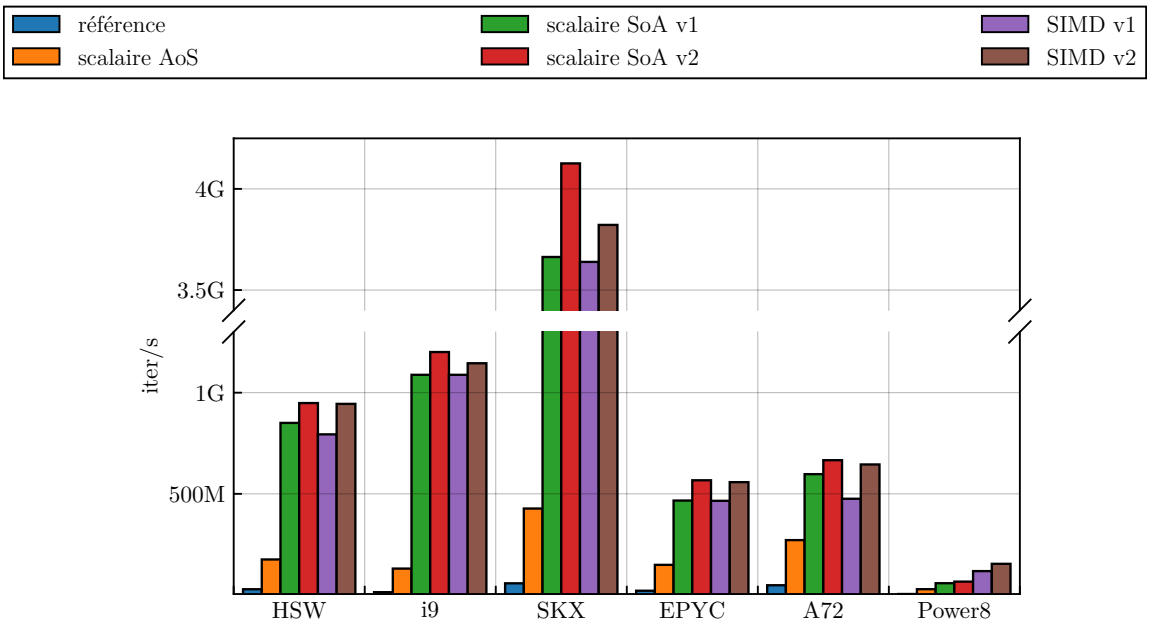
La figure 5.3 présente la vitesse de traitement du filtre de Kalman des différentes machines testées. On remarque rapidement que les vitesses de traitement des différentes machines sont trop différentes pour une lecture aisée. La raison est que les machines elles-mêmes sont très différentes avec un nombre de cœurs variant de 4 à 48, une largeur SIMD de 128 à 512 bits.

La vitesse de traitement normalisée donnée par la figure 5.4 permet de s'affranchir de ce problème, et ainsi de faciliter la lecture et l'analyse. À cause de la normalisation, il n'est pas possible de comparer directement les architectures entre elles.

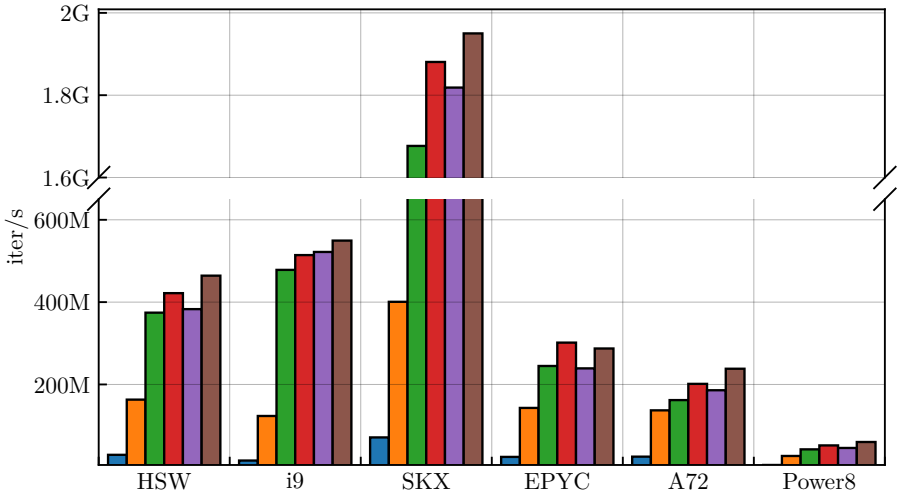
En simple précision (figure 5.4a), les versions SIMD ne sont pas plus rapides que les versions scalaires vectorisées. Ces versions scalaires sont même légèrement meilleures sur la plupart des architectures. L'ordonnancement des instructions et l'allocation de registre sont peut-être impliqués dans cette différence.

Il faut toutefois noter que l'utilisation de `#pragma omp simd` est requise ici pour que les compilateurs vectorisent. Dans le cas contraire, les heuristiques des compilateurs ont tendances à empêcher la vectorisation de boucles aussi longues.

Tout comme dans le cas de la factorisation de Cholesky, les implémentations passent bien à l'échelle avec une efficacité multi-cœur supérieure à 80%.

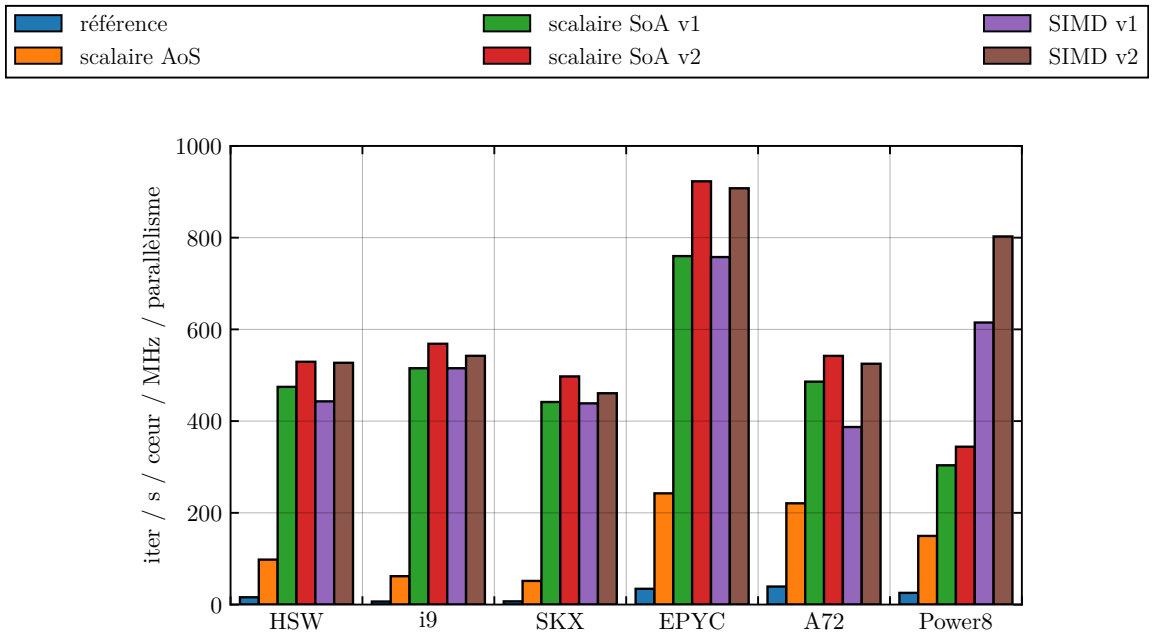


(a) F32

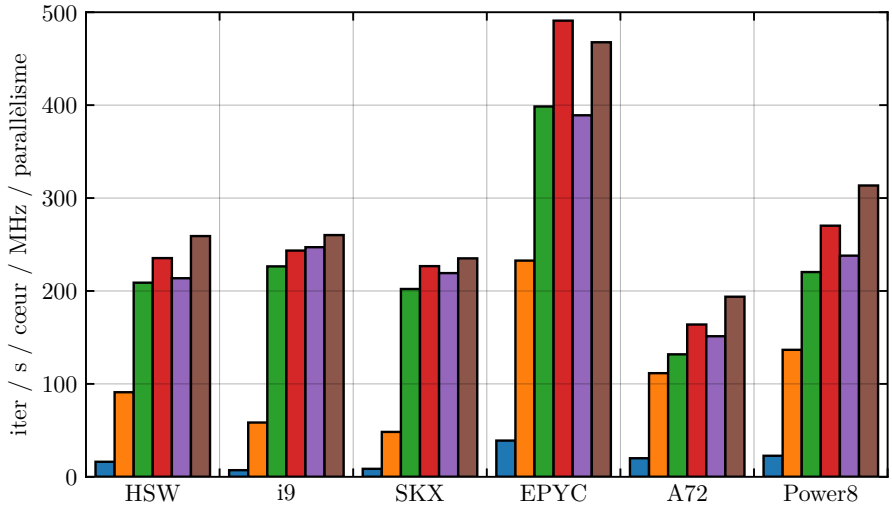


(b) F64

Figure 5.3 – Vitesse de traitement du filtre de Kalman 4×4



(a) F32



(b) F64

Figure 5.4 – Vitesse de traitement normalisée du filtre de Kalman 4×4

Table 5.1 – Nomenclature du filtre de Kalman LHCb

x	$\mathbb{R}^5$		état	“$\left(x, y, \frac{dx}{dz}, \frac{dy}{dz}, \frac{q}{p}\right)$”
x'	$\mathbb{R}^5$		état prédit	
x^{ref}	$\mathbb{R}^5$		état de référence	trajectoire théorique
P	$\mathbb{S}^5_+$		covariance de l'état	
P'	$\mathbb{S}^5_+$		covariance prédite	
A	$\mathbb{R}^{5\times5}$		matrice de transport	évolution du système
b	$\mathbb{R}^5$		vecteur de transport	contrôl
Q	$\mathbb{S}^5_+$		bruit du processus	
H	$\mathbb{R}^{1\times5}$	$\cong \mathbb{R}^5$	projection	
R	$\mathbb{S}^1_+$	$\cong \mathbb{R}^+$	bruit de la mesure	
$\tilde{y}$	$\mathbb{R}^1$	$\cong \mathbb{R}$	résidu	innovation
$\tilde{y}^{\text{ref}}$	$\mathbb{R}^1$	$\cong \mathbb{R}$	résidu de référence	distance à la trajectoire théorique
S	$\mathbb{S}^1_+$	$\cong \mathbb{R}^+$	covariance du résidu	
K	$\mathbb{R}^{5\times1}$	$\cong \mathbb{R}^5$	gain de Kalman	
K'	$\mathbb{R}^{5\times5}$		poids de la passe	
χ^2	$\mathbb{R}^+$		vraisemblance	
$\overset{\square}{\rightarrow}$			$\square$ pour la passe en avant	
$\overset{\square}{\leftarrow}$			$\square$ pour la passe en arrière	
$\overset{\square}{\leftrightarrow}$			$\square$ pour la passe de lissage	

5.4 Filtre de Kalman spécifique à LHCb

Au sein de LHCb, le filtre de Kalman est plus complexe que celui vu précédemment. Tout d'abord, le système dont on veut suivre l'évolution est la trajectoire d'une particule. Ce système est non-linéaire de par la présence d'un champs magnétique et des matériaux qui peuvent dévier la particule. Il faut donc linéariser les équations, ce qui a pour conséquence que chaque point de la trajectoire aura des paramètres différents (A, Q, H et R). De plus, chaque point correspondant à une partie du détecteur, le nombre total de point est fixe, de l'ordre de 20. On peut donc appliquer le filtre dans les deux sens pour affiner la mesure. Un tel filtre a été décrit par Rudolf Früwirth [105].

5.4.1 Algorithme

On distingue trois passes : en avant (*forward*), en arrière (*backward*) et le lissage (*smoothing*). La passe en avant correspond au filtre de Kalman classique. La passe en arrière est presque identique à la passe en avant, à l'exception que les équations de prédictions sont inversées. La passe de lissage fait la moyenne des deux passes en pondérant les états par les covariances. L'algorithme 5.4 rassemble ces trois étapes. À noter que les passes en arrière et de lissage peuvent être entrelacées afin d'augmenter la localité des accès mémoires.

Algorithme 5.4 Filtre de Kalman LHCb (noyaux de calculs : algorithme 5.5)

// avant
1 $\underset{\rightarrow}{x}_0, \underset{\rightarrow}{P}_0 \leftarrow$ default
2 **pour** $i = 1 : n$ **faire**
3 $\quad \underset{\rightarrow}{x}'_i, \underset{\rightarrow}{P}'_i \leftarrow \text{predict forward}\Big(\underset{\rightarrow}{x}_{i-1}, \underset{\rightarrow}{P}_{i-1}, b_i, A_i, Q_i\Big)$
4 $\quad \underset{\rightarrow}{\tilde{y}}_i, \underset{\rightarrow}{S}_i, \underset{\rightarrow}{\chi}^2_i \leftarrow \text{update residuals}\Big(\underset{\rightarrow}{x}'_i, \underset{\rightarrow}{P}'_i, H_i, R_i, \tilde{y}^{\text{ref}}_i, x^{\text{ref}}_i\Big)$
5 $\quad \underset{\rightarrow}{x}_i, \underset{\rightarrow}{P}_i \leftarrow \text{update}\Big(\underset{\rightarrow}{x}'_i, \underset{\rightarrow}{P}'_i, H_i, \underset{\rightarrow}{\tilde{y}}_i, \underset{\rightarrow}{S}_i\Big)$

// arrière
6 $\underset{\leftarrow}{x}_n, \underset{\leftarrow}{P}_n \leftarrow \underset{\rightarrow}{x}_n, \underset{\rightarrow}{P}_n$
7 **pour** $i = n-1 : 0$ **faire**
8 $\quad \underset{\leftarrow}{x}'_i, \underset{\leftarrow}{P}'_i \leftarrow \text{predict backward}\Big(\underset{\leftarrow}{x}_{i+1}, \underset{\leftarrow}{P}_{i+1}, b_i, A_i^{-1}, Q_i\Big)$
9 $\quad \underset{\leftarrow}{\tilde{y}}_i, \underset{\leftarrow}{S}_i, \underset{\leftarrow}{\chi}^2_i \leftarrow \text{update residuals}\Big(\underset{\leftarrow}{x}'_i, \underset{\leftarrow}{P}'_i, H_i, R_i, \tilde{y}^{\text{ref}}_i, x^{\text{ref}}_i\Big)$
10 $\quad \underset{\leftarrow}{x}_i, \underset{\leftarrow}{P}_i \leftarrow \text{update}\Big(\underset{\leftarrow}{x}'_i, \underset{\leftarrow}{P}'_i, H_i, \underset{\leftarrow}{\tilde{y}}_i, \underset{\leftarrow}{S}_i\Big)$

// lissage
11 **pour** $i = n-1 : 0$ **faire**
12 $\quad \underset{\leftrightarrow}{x}_i, \underset{\leftrightarrow}{P}_i \leftarrow \text{smooth}\Big(\underset{\rightarrow}{x}_i, \underset{\leftarrow}{x}'_i, \underset{\rightarrow}{P}_i, \underset{\leftarrow}{P}'_i\Big)$
13 $\quad \underset{\leftrightarrow}{\tilde{y}}_i, \underset{\leftrightarrow}{S}_i, \underset{\leftrightarrow}{\chi}^2_i \leftarrow \text{update residuals}\Big(\underset{\leftrightarrow}{x}_i, \underset{\leftrightarrow}{P}_i, H_i, R_i, \tilde{y}^{\text{ref}}_i, x^{\text{ref}}_i\Big)$

La nomenclature utilisée est décrite par le tableau 5.1. On y remarque ainsi la présence de grandeurs scalaires (R, $\tilde{y}$, $\tilde{y}^{\text{ref}}$, S et χ^2) et de matrices symétriques définies positives 5×5 (P, P' et Q). L'état x et sa covariance P sont différents pour chaque passe. On distinguera ainsi l'état de la passe en avant $\underset{\rightarrow}{x}$ de la passe en arrière $\underset{\leftarrow}{x}$ et de la passe de lissage $\underset{\leftrightarrow}{x}$.

Certaines parties sont identiques à plusieures passes. On peut donc factoriser l'implémentation des passes et définir plusieurs noyaux de calculs (algorithme 5.5). On distingue ainsi les noyaux suivants :

— *predict forward* : prédit le nouvel état lors de la passe en avant,
— *predict backward* : prédit le nouvel état lors de la passe en arrière,
— *update residuals* : calcul le résidu et la vraisemblance (commun à toutes les passes),
— *update* : mets à jour l'état en fonction du résidu (passes en avant et en arrière),
— *smooth* : lisse l'état en utilisant les états des passes en avant et en arrière.

Le calcul du résidu lors de la passe de lissage est nécessaire pour le calcul de la vraisemblance.

Algorithme 5.5 Filtre de Kalman LHCb : noyaux de calcul

(a) *predict forward* : 425 ops

1 $x' \leftarrow A\,x + b$
2 $P' \leftarrow A\,P\,A^{\mathsf{T}} + Q$

(b) *predict backward* : 440 ops

1 $x' \leftarrow A^{-1}\,(x - b)$
2 $P' \leftarrow A^{-1}\,(P - Q)\,A^{-1\mathsf{T}}$
// L'inverse de A est précalculée

(c) *update residuals* : 73 ops

1 $\tilde{y} \leftarrow \tilde{y}^{\text{ref}} + H\,\left(x^{\text{ref}} - x'\right)$
2 $S \leftarrow H\,P'\,H^{\mathsf{T}} + R$
3 $\chi^2 \leftarrow \tilde{y}^{\mathsf{T}}\,S\,\tilde{y}$

(d) *update* : 245 ops

1 $K \leftarrow P'\,H^{\mathsf{T}}\,S^{-1}$
2 $x \leftarrow x' + K\,\tilde{y}$
3 $P \leftarrow (I - K\,H)\,P'$

(e) *smooth* : 585 ops

1 $K' \leftarrow \underset{\rightarrow}{P}\,\left(\underset{\rightarrow}{P} + \underset{\leftarrow}{P'}\right)^{-1}$
2 $\underset{\leftrightarrow}{x} \leftarrow \underset{\rightarrow}{x} + K'\,\left(\underset{\leftarrow}{x'} - \underset{\rightarrow}{x}\right)$
3 $\underset{\leftrightarrow}{P} \leftarrow K'\,\underset{\leftarrow}{P'}$

(f) RTS (Rauch-Tung-Striebel) : 1085 ops

1 $C \leftarrow \underset{\rightarrow}{P}\,A^{\mathsf{T}}\underset{\rightarrow}{P'}^{-1}$
2 $\underset{\leftrightarrow}{x} \leftarrow \underset{\rightarrow}{x} + C\,\left(\underset{\leftrightarrow}{x} - \underset{\rightarrow}{x'}\right)$
3 $\underset{\leftrightarrow}{P} \leftarrow \underset{\rightarrow}{P} + C\,\left(\underset{\leftrightarrow}{P} - \underset{\rightarrow}{P'}\right)\,C^{\mathsf{T}}$

5.4.2 Algorithmes alternatifs

Cet algorithme peut être modifié pour diminuer les calculs, ou améliorer la précision [106]. Il est possible de *fusionner* les passes en arrière et de lissage de la manière suivante : le nouvel état est prédit à partir de l'état lissé directement (algorithme 5.6). Il est ainsi possible d'éviter la mise à jour de l'état de la passe en arrière (*update residuals* et *update*), celle-ci étant remplacée par le lissage nécessaire quoiqu'il arrive. Cette version alternative du filtre donne des résultats différents, mais nécessite moins de calcul (tableau 5.2). De plus, bien que les résultats soient différent, ceux-ci devraient être plus proches de la réalité car plus d'information est propagée lors de la passe en arrière. Cette dernière affirmation n'a cependant pas été vérifiée par manque de temps.

Il existe dans la littérature une extension au filtre de Kalman qui permet de lisser avec une passe en arrière. Il s'agit du lissage de Rauch-Tung-Striebel [107]. Ce lissage nécessite deux passes : la première est identique à la passe en avant du filtre de Kalman. La deuxième passe est en arrière et calcul l'état lissé directement à partir des états filtrés et prédits de la passe en avant, ainsi que de l'état lissé précédent. L'algorithme 5.7 présente l'application de ce filtre à notre système.

Il existe également d'autres formulations alternatives qui n'ont pas été explorées ici. On peut citer le filtre d'information [108] (*Information filter*) qui consiste à faire les calculs

Algorithme 5.6 Filtre de Kalman LHCb alternatif : *backward* et *smoother* fusionnés

// avant

1 $\underset{\rightarrow}{x}_0, \underset{\rightarrow}{P}_0 \leftarrow$ default

2 **pour** $i = 1 : n$ **faire**

3 $\quad \underset{\rightarrow}{x}'_i, \underset{\rightarrow}{P}'_i \leftarrow$ predict forward$\left(\underset{\rightarrow}{x}_{i-1}, \underset{\rightarrow}{P}_{i-1}, b_i, A_i, Q_i\right)$

4 $\quad \underset{\rightarrow}{\tilde{y}}_i, \underset{\rightarrow}{S}_i, \underset{\rightarrow}{\chi}^2_i \leftarrow$ update residuals$\left(\underset{\rightarrow}{x}'_i, \underset{\rightarrow}{P}'_i, H_i, R_i, \tilde{y}^{\text{ref}}_i, x^{\text{ref}}_i\right)$

5 $\quad \underset{\rightarrow}{x}_i, \underset{\rightarrow}{P}_i \leftarrow$ update$\left(\underset{\rightarrow}{x}'_i, \underset{\rightarrow}{P}'_i, H_i, \underset{\rightarrow}{\tilde{y}}_i, \underset{\rightarrow}{S}_i\right)$

// arrière (lissage)

6 $\underset{\leftrightarrow}{x}_n, \underset{\leftrightarrow}{P}_n \leftarrow \underset{\rightarrow}{x}_n, \underset{\rightarrow}{P}_n$

7 **pour** $i = n - 1 : 0$ **faire**

8 $\quad \underset{\leftarrow}{x}'_i, \underset{\leftarrow}{P}'_i \leftarrow$ predict backward$\left(\underset{\leftrightarrow}{x}_{i+1}, \underset{\leftrightarrow}{P}_{i+1}, b_i, A_i^{-1}, Q_i\right)$

9 $\quad \underset{\leftrightarrow}{x}_i, \underset{\leftrightarrow}{P}_i \leftarrow$ smooth$\left(\underset{\rightarrow}{x}_i, \underset{\leftarrow}{x}'_i, \underset{\rightarrow}{P}_i, \underset{\leftarrow}{P}'_i\right)$

10 $\quad \underset{\leftrightarrow}{\tilde{y}}_i, \underset{\leftrightarrow}{S}_i, \underset{\leftrightarrow}{\chi}^2_i \leftarrow$ update residuals$\left(\underset{\leftrightarrow}{x}_i, \underset{\leftrightarrow}{P}_i, H_i, R_i, \tilde{y}^{\text{ref}}_i, x^{\text{ref}}_i\right)$

Algorithme 5.7 Filtre de Kalman LHCb alternatif : Rauch-Tung-Striebel (algorithme 5.5f)

// avant

1 $\underset{\rightarrow}{x}_0, \underset{\rightarrow}{P}_0 \leftarrow$ default

2 **pour** $i = 1 : n$ **faire**

3 $\quad \underset{\rightarrow}{x}'_i, \underset{\rightarrow}{P}'_i \leftarrow$ predict forward$\left(\underset{\rightarrow}{x}_{i-1}, \underset{\rightarrow}{P}_{i-1}, b_i, A_i, Q_i\right)$

4 $\quad \underset{\rightarrow}{\tilde{y}}_i, \underset{\rightarrow}{S}_i, \underset{\rightarrow}{\chi}^2_i \leftarrow$ update residuals$\left(\underset{\rightarrow}{x}'_i, \underset{\rightarrow}{P}'_i, H_i, R_i, \tilde{y}^{\text{ref}}_i, x^{\text{ref}}_i\right)$

5 $\quad \underset{\rightarrow}{x}_i, \underset{\rightarrow}{P}_i \leftarrow$ update$\left(\underset{\rightarrow}{x}'_i, \underset{\rightarrow}{P}'_i, H_i, \underset{\rightarrow}{\tilde{y}}_i, \underset{\rightarrow}{S}_i\right)$

// arrière (lissage)

6 $\underset{\leftrightarrow}{x}_n, \underset{\leftrightarrow}{P}_n \leftarrow \underset{\rightarrow}{x}_n, \underset{\rightarrow}{P}_n$

7 **pour** $i = n - 1 : 0$ **faire**

8 $\quad \underset{\leftrightarrow}{x}_i, \underset{\leftrightarrow}{P}_i \leftarrow$ RTS$\left(\underset{\rightarrow}{x}_i, \underset{\rightarrow}{x}'_{i+1}, \underset{\leftrightarrow}{x}_{i+1}, \underset{\rightarrow}{P}_i, \underset{\rightarrow}{P}'_{i+1}, \underset{\leftrightarrow}{P}_{i+1}, A_{i+1}\right)$

9 $\quad \underset{\leftrightarrow}{\tilde{y}}_i, \underset{\leftrightarrow}{S}_i, \underset{\leftrightarrow}{\chi}^2_i \leftarrow$ update residuals$\left(\underset{\leftrightarrow}{x}_i, \underset{\leftrightarrow}{P}_i, H_i, R_i, \tilde{y}^{\text{ref}}_i, x^{\text{ref}}_i\right)$

Table 5.2 – Complexité du filtre de Kalman LHCb

Version	Opération/point	Accès mémoire/point	IA
3 passes	2159	244	8.8
fusionnées	1841	235	7.8
RTS	1901	257	7.4

sur les matrices inverses des covariances. On peut également faire les calculs sur les formes factorisées des matrices de covariances : on a alors un filtre "racine carrée" [109] (*Square root filter*).

5.4.3 Résultats

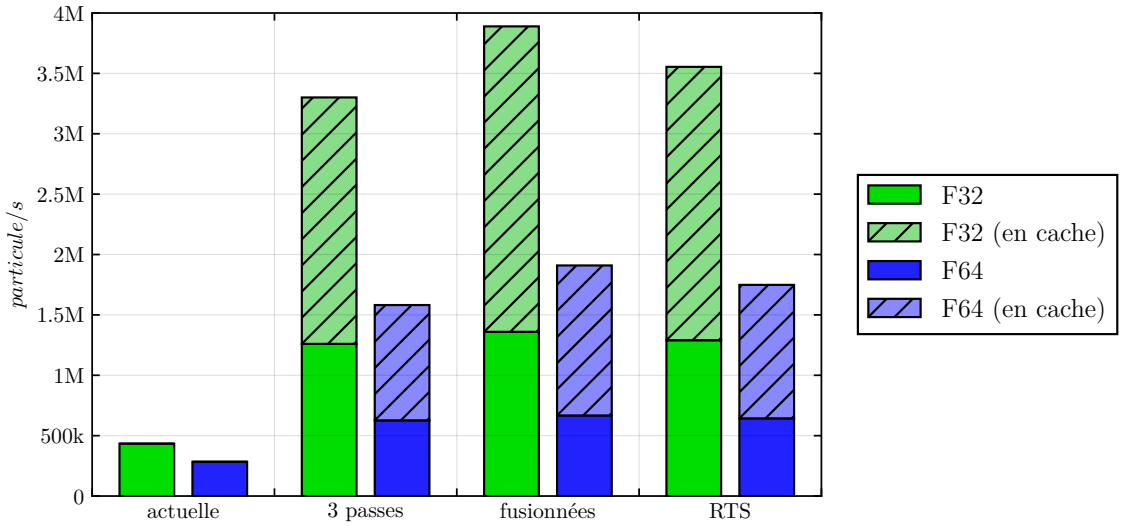
Le protocole pour mesurer la vitesse de traitement du filtre de Kalman LHCb reste le même. Nous ne nous intéresseront ici qu'à la meilleure implémentation de chaque version.

On distingue ainsi 4 versions :
- actuelle : correspond à une version extraite du code de production [110, 111],
- 3 passes : la même version réimplémentée avec toutes les optimisations vues jusqu'à présent (algorithme 5.4),
- fusionnées : les passes en arrière (*backward*) et de lissage (*smoothing*) sont fusionnées (algorithme 5.6),
- RTS (Rauch-Tung-Stribel) : les passes en arrière et de lissage sont remplacées par l'algorithme de Rauch-Tung-Striebel (algorithme 5.7).

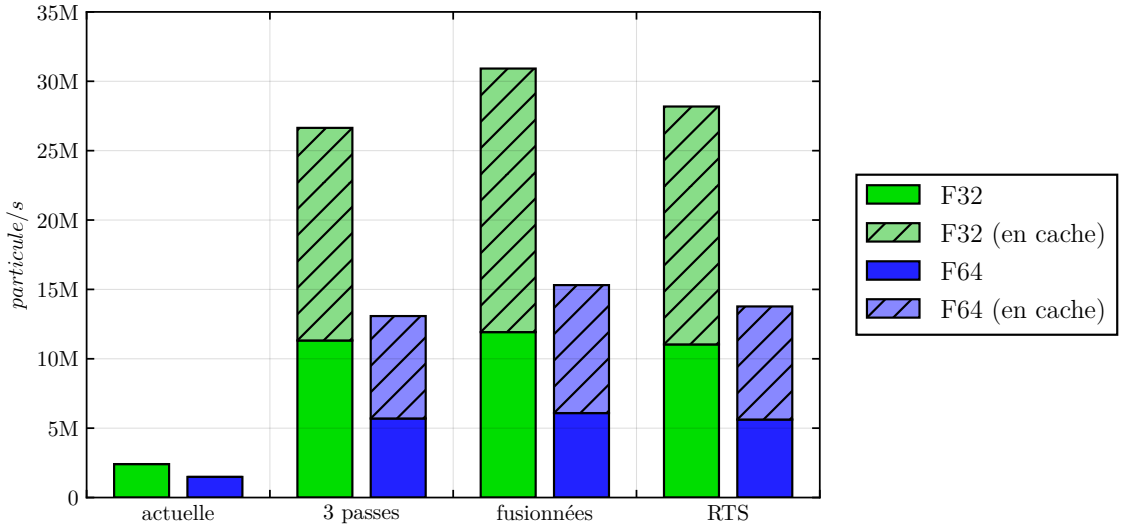
Ces versions, y compris la version "actuelle", ne sont pas équivalente au code de production. Elles ne présentent pas le calcul des matrices de bruits et de transitions, ces calculs étant faits en amont. Le code extrait de la version actuelle ne permet pas de mesurer la vitesse de traitement lorsque les données tiennent en cache.

La vitesse de traitement de chaque version est reportée par la figure 5.5. On y remarque que les optimisations apportées à la version actuelle permettent d'accélérer la vitesse de traitement d'un facteur compris entre $\times 2.2$ et $\times 2.9$ à précision égale. Ce facteur grimpe entre $\times 3.8$ et $\times 4.7$ lorsque le code est exécuté sur tous les cœurs.

Fusionner les passes en arrière et de lissage permet d'accélérer le code entre $\times 1.05$ et $\times 1.20$ par rapport à la version en 3 passes. De la même manière, Rauch-Tung-Stribel permet d'accélérer entre $\times 1.02$ et $\times 1.10$, mais uniquement sur un seul cœur. Sur tous les cœurs, cette version est légèrement plus lente que la version en 3 passes lorsque les données ne tiennnent pas en cache. Il faut cependant noter que cet algorithme devrait donner des résultats moins biaisés, et que l'absence de l'inversion de la matrice de transition A devrait fournir une meilleure stabilité numérique. Ces pistes n'ont cependant pas pu être explorées par manque de temps.



(a) Mono cœur



(b) OpenMP

Figure 5.5 – Vitesse de traitement du filtre de Kalman LHCb (i9)

Table 5.3 – Efficacité parallèle du filtre de kalman LHCb (i9, 10 cœurs)

Version	en cache		en mémoire externe	
	F32	F64	F32	F64
actuelle	N/A	N/A	×5.6	×5.2
3 passes	×8.1	×8.3	×9.0	×9.1
fusionnées	×7.9	×8.0	×8.8	×9.1
RTS	×7.9	×7.9	×8.6	×8.7

On remarque également que la vitesse en simple précision est au minimum ×1.95 celle en double précision. La version actuelle en revanche présente une accélération de ×1.5 entre les deux précisions de calculs. Enfin, la vitesse de traitement lorsque les données tiennent en cache est ×2.5 supérieure par rapport aux données en mémoire externe. Bien que l'intensité arithmétique soit bonne ($\simeq 8$), le rapport entre la puissance de calcul et la bande passante de la machine i9 est tellement haut ($\simeq 135$) que le problème reste *memory bound* sur cette machine.

Les résultats mono-cœur et multi-cœur sont très similaires. Le passage à l'échelle est bon avec une efficacité parrallèle $> 80\%$ (tableau 5.3). On remarque également que la version actuelle passe moins bien à l'échelle avec une efficacité parallèle entre 50% et 60%.

On peut voir la puissance de calcul et la bande passante générées par les différentes versions dans le tableau 5.4. La bande passante générée est supérieure à la bande passante de la mémoire externe et même du cache L3 : il y a réutilisation des données d'une itération à l'autre, ainsi qu'entre les différentes passes.

Comparaison avec l'État de l'Art. Bien que chaque expérience en Physique des Hautes Énergies ait des caractéristiques différentes, et donc une implémentation du filtre de Kalman différente, il est possible dans une certaine mesure de comparer nos résultats avec d'autres filtres de Kalman. Le tableau 5.5 compare ainsi la vitesse de traitement de notre filtre de Kalman avec les filtres de CMS [112] et CBM [113]. Les vitesses de traitement des filtres de CMS et CBM sont estimées à partir de leur publication respective.

On constate ainsi que notre meilleure version en simple précision est ×2.3 plus rapide que le filtre de CMS en simple précision à parallèlisme identique, tandis que la version actuelle est ×1.4 plus lente. Lorsque les données sont en caches, notre meilleure version est ×6.5 plus rapide que le filtre de CMS.

Notre meilleure version est ×11 plus rapide que le filtre CBM lorsque les données sont en cache. Le filtre CBM est toutefois plus difficile à comparer du fait d'un parallèlisme plus faible.

Les deux filtres CMS et CBM n'ont cependant qu'une seule passe tandis que notre algorithme en possède au moins deux (suivant les versions).

Table 5.4 – Bande passante et puissance de calcul générées par le filtre de Kalman LHCb (i9)

			actuelle*	3 passes	fusionnées	RTS
Gflops	cache	F32	N/A	1090	1080	1018
	cache	F64	N/A	536	536	497
	RAM	F32	98.9	464	417	399
	RAM	F64	61.0	234	213	203
Go/s	cache	F32	N/A	520	581	579
	cache	F64	N/A	511	576	566
	RAM	F32	47	221	224	227
	RAM	F64	58	222	229	231

* : La version actuelle n'est limitée ni par la puissance de calcul, ni par la bande passante de la mémoire

Table 5.5 – Comparaison du filtre de Kalman LHCb avec l'état de l'art

Version			SIMD	#Passes	cycle/point F32	cycle/point F64	
LHCb	actuelle	(5×5)	AVX512	3*	436	667	RAM
	3 passes	(5×5)	AVX512	3*	150	302	
	fusionnées	(5×5)	AVX512	2*	139	284	
	RTS	(5×5)	AVX512	2*	147	294	
LHCb	3 passes	(5×5)	AVX512	3*	57	120	cache
	fusionnées	(5×5)	AVX512	2*	49	99	
	RTS	(5×5)	AVX512	2*	53	108	
	CMS	(3→6 ?)	AVX	1	520	N/A	
	CMS	(3→6 ?)	KNCNI	1	320	N/A	
	CBM	(5×5)	SSE2	1	560	N/A	

∗ correspond à l'équivalent de 3 passes : en avant, en arrière et de lissage

5.5 Synthèse

Dans cette partie, nous avons vu le filtre de Kalman, un algorithme important dans beacuoup de domaines. Parmi les transformations testées, les plus bénéfiques sont le déroulage total (de $\times 3.4$ à $\times 6.8$), le passage à *AoSoA* (de 0.62 à 1.4 fois le cardinal du SIMD), l'accès triangulaire aux matrices symétriques ($\times 1.2$) et les tranformations mathématiques ($\times 1.2$). Les autres optimisations (la racine carrée rapide, le SIMD avec les *intrinsics*, dérouler–entrelacer ou le réordonnancement) ont un effet sur la vitesse de traitement négligeable voire néfaste dans de rares cas. La version la plus performante permet de calculer $4 \cdot 10^9$ itérations du filtre de Kalman 4×4 par seconde sur un puissant serveur bi-CPU (SKX).

Le filtre de Kalman LHCb est plus difficile et nécessite 3 passes (en avant, en arrière, et lissage). Les deux dernières passes peuvent être fusionnées afin d'économiser des calculs et des accès mémoires. Les autres optimisations sont également appliquées. Le code obtenu est de $\times 2.2$ à $\times 2.9$ plus rapide que la version actuelle utilisant déjà le SIMD. Il passe également mieux à l'échelle avec une efficacité multi-cœur $> 85\%$ contre $\simeq 55\%$ pour la version actuelle. De plus, cette nouvelle version est au moins $\times 2.3$ plus rapide que les filtres de Kalman d'autres expériences de physique des particules à parallèlisme égal, malgré un nombre de passes plus grand.

Chapitre 6

Conclusion

Tout au long de cette thèse [114–116], nous avons étudié des problèmes d'algèbre linéaire de petite dimension (typiquement de 2×2 à 5×5) utilisés dans la reconstruction de la trajectoire des particules pour l'expérience LHCb. Les bibliothèques d'algèbre linéaire telles que Eigen, Magma ou la MKL ne sont pas optimisées pour des petites matrices. La reconstruction LHCb, tout comme d'autres domaines tels que la vision par ordinateur, nécessite le traitement de beaucoup de ces petites matrices. Il est donc crucial d'optimiser leur vitesse de traitement afin d'accélérer la vitesse de la reconstruction complète.

Nous avons ainsi utilisé et combiné plusieurs transformations connues telles que le changement de l'agencement mémoire de *AoS* vers *AoSoA* plus adapté au SIMD, le déroulage avec entrelacement permettant de s'affranchir de la latence des instructions, le déroulage total de boucle et la scalarisation diminuant le nombre d'accès mémoires.

À ces transformations s'ajoutent des transformations moins usuelles comme la racine carré inverse rapide ou le réordonnancement source à source du code. Pour faciliter l'écriture de ces transformations, mais également dans le but d'avoir un code portable, nous avons écrit un générateur de code en Python se basant sur le moteur de *templates* Jinja2.

Nous avons testé ces transformations et analysé leur impact sur la vitesse de traitement d'algorithmes simples de la séquence de reconstruction LHCb. Le traitement par lot (*batch*) en *SoA* est crucial pour maximiser la vitesse de traitements de ces problèmes à faible dimension.

Nous avons également analysé la distribution de la vitesse de traitement de ces fonctions afin de s'assurer de la validité de notre méthode de mesure de temps. Ainsi, prendre le temps minimum sur plusieurs exécutions permet de s'affranchir des aléas de la machine et des interruptions régulières.

Nous nous sommes également servis de ces exemples simples pour faire une analyse de la précision des résultats en fonction de la précision de calcul. Le nombre de bits incorrects

du résultat est indépendant de la précision de calcul.

Nous avons alors implémenté ces transformations dans le but d'accélérer la factorisation de Cholesky de petites matrices (jusqu'à 12×12). La vitesse de traitement de la factorisation de Cholesky plafonne sans l'utilisation du calcul rapide de la racine carrée inverse.

Nous avons ainsi obtenu, en simple précision sur un processeur Xeon Haswell comparé au code sans transformations, une accélération de ×33 en 3×3 et de ×13 en 12×12. Notre version est ainsi ×10 plus rapide que la MKL en 3×3 et ×3 plus rapide en 12×12 dans les mêmes conditions. Le code passe également bien à l'échelle avec une efficacité parallèle $> 80\%$.

Enfin, nous avons étudié et accéléré le filtre de Kalman généraliste. Nous avons ainsi obtenu une accélération de ×90 sur l'implémentation 4×4 simple précision sur un serveur bi-CPU Skylake par rapport au code sans transformation. Cette machine est alors capable de traiter $4 \cdot 10^9$ points par seconde.

Le filtre de Kalman utilisé au sein de LHCb (représentant presque la moitié de la reconstruction) est plus complexe et nécessite 3 passes (en avant, en arrière et lissage). La fusion des deux dernières passes couplée aux autres transformations permet d'obtenir un code de ×2.2 à ×2.9 plus rapide que la version actuelle utilisant déjà le SIMD, et au moins ×2.3 plus rapide que les filtres de Kalman utilisés par d'autres expériences de physique des particules à parallèlisme égal. Il passe également mieux à l'échelle avec une efficacité parallèle $> 85\%$ contre $\simeq 55\%$ pour la version actuelle.

Perspectives

Les premiers travaux futurs considérés sont l'intégration de ces codes au sein du *framework* LHCb. Cette intégration peut se découper en deux étapes indépendantes : la création d'une bibliothèque pour la factorisation de Cholesky en faible dimension pouvant être réutilisée telle quelle au sein de la reconstruction, et le portage de notre implémentation du filtre de Kalman pour le *framework* LHCb directement.

Les GPU actuels ont des latences de tranferts plus faible qu'au début de la thèse. Ces derniers présentent donc un intérêt nouveau pour le traitement de problèmes à faible dimension. Nous envisageons donc de poursuivre et d'étendre ces travaux aux GPU.

Enfin, les travaux de cette thèse peuvent être poursuivis afin de créer un guide présentant différentes transformations et comment les appliquer dans le but d'accélerer des problèmes de faible dimension.

Bibliographie

[1] I. Bediaga, H. Chanal, P. Hopchev, S. Cadeddu, S. Stoica, M. Calvo Gomez, S. T'Jampens, I. Machikhiliyan, Z. Guzik, A. Alves Jr, *et al.*, "Framework TDR for the LHCb upgrade : Technical design report," tech. rep., LHCb-TDR-012, 2012.

[2] R. Aaij, M. Fontana, R. Le Gac, E. A. Zacharjasz, R. Schwemmer, C. Fitzpatrick, J. Albrecht, L. Grillo, T. Szumlak, H. Yin, *et al.*, "Upgrade trigger : Biannual performance update," tech. rep., 2017.

[3] G. Guennebaud, B. Jacob, *et al.*, "Eigen v3." http ://eigen.tuxfamily.org/, 2016.

[4] S. Tomov, R. Nath, P. Du, and J. Dongarra, "Magma, matrix algebra on gpu and multicore architectures." http ://icl.cs.utk.edu/magma/.

[5] MKL, "Intel(R) math kernel library." `https://software.intel.com/en-us/intel-mkl`.

[6] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. Dongarra, "High-performance matrix-matrix multiplications of very small matrices," in *European Conference on Parallel Processing*, pp. 659–671, Springer International Publishing, 2016.

[7] J. Shin, M. W. Hall, J. Chame, C. Chen, and P. D. Hovland, "Autotuning and specialization : Speeding up matrix multiply for small matrices with compiler technology," in *Software Automatic Tuning*, pp. 353–370, Springer, 2011.

[8] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "LIBXSMM : accelerating small matrix multiplications by runtime code generation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 84, IEEE Press, 2016.

[9] X. Tian, H. Saito, S. V. Preis, E. N. Garcia, S. S. Kozhukhov, M. Masten, A. G. Cherkasov, and N. Panchenko, "Effective SIMD vectorization for Intel Xeon Phi coprocessors," *Scientific Programming*, vol. 2015, pp. 1–14, Jan. 2015.

[10] D. G. Spampinato and M. Püschel, "A basic linear algebra compiler," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, p. 23, ACM, 2014.

[11] A. R. M. y Terán, L. Lacassagne, A. H. Zahraee, and M. Gouiffes, "Real-time covariance tracking algorithm for embedded systems," in *Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on*, pp. 104–111, IEEE, 2013.

[12] F. Laguzet, A. Romero, M. Gouiffès, L. Lacassagne, and D. Etiemble, "Color tracking with contextual switching : real-time implementation on CPU," *Journal of Real-Time Image Processing*, vol. 10, no. 2, pp. 403–422, 2015.

[13] R. Allen and K. Kennedy, eds., *Optimizing compilers for modern architectures : a dependence-based approach*, ch. 8,9,11. Morgan Kaufmann, 2002.

[14] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache, "Facilitating the search for compositions of program transformations," in *Proceedings of the 19th annual international conference on Supercomputing*, pp. 151–160, ACM, 2005.

[15] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, Sept 1972.

[16] P. Estérie, J. Falcou, M. Gaunard, and J.-T. Lapresté, "Boost.SIMD : Generic programming for portable SIMDization," in *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, (New York, NY, USA), pp. 1–8, ACM, 2014.

[17] T. Ewart, F. Delalondre, and F. Schürmann, "Cyme : A library maximizing SIMD computation on user-defined containers," in *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ISC 2014, (New York, NY, USA), pp. 440–449, Springer-Verlag New York, Inc., 2014.

[18] libsimdpp, "Header-only zero-overhead c++ wrapper for simd intrinsics of multiple instruction sets." `https://github.com/p12tic/libsimdpp`, 2017. commit : 7c2b867.

[19] A. Cassagne, B. Le Gal, C. Leroux, O. Aumage, and D. Barthou, "An efficient, portable and generic library for successive cancellation decoding of polar codes," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 303–317, Springer, 2015.

[20] P. Karpiński and J. McDonald, "A high-performance portable abstract interface for explicit SIMD vectorization," in *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'17, (New York, NY, USA), pp. 21–28, ACM, 2017.

[21] M. Kretz and V. Lindenstruth, "Vc : A C++ library for explicit vectorization," *Software : Practice and Experience*, vol. 42, no. 11, pp. 1409–1430, 2012.

[22] A. Fog, "C++ vector class library." `http://agner.org/optimize/vectorclass.zip`, 2017. accessed version : 2017-07-27.

[23] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin, "Task parallelism and data distribution : An overview of explicit parallel programming languages," in *International*

Workshop on Languages and Compilers for Parallel Computing, pp. 174–189, Springer, 2012.

[24] M. Pharr and W. R. Mark, "ispc : A SPMD compiler for high-performance CPU programming," in *Innovative Parallel Computing (InPar), 2012*, pp. 1–13, IEEE, 2012.

[25] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC–first experiences with real-world applications," in *European Conference on Parallel Processing*, pp. 859–870, Springer, 2012.

[26] A. Munshi, "The OpenCL specification," in *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pp. 1–314, IEEE, 2009.

[27] L. Dagum and R. Menon, "OpenMP : an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[28] "IEEE standard for information technology–portable operating system interface (POSIX(R)) base specifications, issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, Jan 2018.

[29] Intel Corporation, *Intel® C++ Compiler 18.0 Developer Guide and Reference*, May 2018. accessed version : 2018-06-29.

[30] "Programming languages – C," standard, International Organization for Standardization, Geneva, CH, Dec. 2011.

[31] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes, "The ILLIAC IV computer," *IEEE Transactions on computers*, vol. 100, no. 8, pp. 746–757, 1968.

[32] J. Watson, "The Texas Instruments Advanced Scientific Computer," in *Managing Requirements Knowledge, International Workshop on(AFIPS)*, vol. 00, p. 221, 12 1899.

[33] C. J. Purcell, "The Control Data STAR-100 : performance measurements," in *Proceedings of the May 6-10, 1974, national computer conference and exposition*, pp. 385–387, ACM, 1974.

[34] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[35] S. F. Reddaway, "DAP–a distributed array processor," in *ACM SIGARCH Computer Architecture News*, vol. 2, pp. 61–65, ACM, 1973.

[36] W. D. Hillis, *The Connection Machine*. MIT press, 1989.

[37] R. J. Fisher and H. G. Dietz, "Compiling for SIMD within a register," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 290–305, Springer, 1998.

[38] *IEEE standard for binary floating-point arithmetic*. New York : Institute of Electrical and Electronics Engineers, 1985. Note : Standard 754–1985.

[39] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, *et al.*, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[40] NEC, "NEC SX-Aurora TSUBASA - Vector Engine." `https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html`, 2018. accessed version : 2018-10-10.

[41] D. M. Carlson, D. C. Sullivan, R. E. Bach, and D. R. Resnick, "The ETA10 liquid-nitrogen-cooled supercomputer system," *IEEE Transactions on Electron Devices*, vol. 36, no. 8, pp. 1404–1413, 1989.

[42] T. Watanabe, "NEC SX-3 supercomputer system," in *Supercomputers and Their Performance in Computational Fluid Dynamics*, pp. 63–75, Springer, 1993.

[43] J. Palmer and G. Steele, "Connection Machine model CM-5 system overview," in *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pp. 474–483, IEEE, 1992.

[44] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark : past, present and future," *Concurrency and Computation : practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[45] J. D. McCalpin, "A survey of memory bandwidth and machine balance in current high performance computers," *IEEE TCCA Newsletter*, vol. 19, p. 25, 1995.

[46] J. Abel, K. Balasubramanian, M. Bargeron, T. Craver, and M. Phlipot, "Applications tuning for streaming SIMD extensions," *Intel Technology Journal*, vol. 2, 1999.

[47] U. Drepper, "What every programmer should know about memory," *Red Hat, Inc*, vol. 11, p. 2007, 2007.

[48] R. Leißa, I. Haffner, and S. Hack, "Sierra : A SIMD extension for C++," in *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP '14, (New York, NY, USA), pp. 17–24, ACM, 2014.

[49] D. Callahan, S. Carr, and K. Kennedy, "Improving register allocation for subscripted variables," *ACM Sigplan Notices*, vol. 25, no. 6, pp. 53–65, 1990.

[50] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization : An overview of the PIPS project," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, pp. 143–150, ACM, 2014.

[51] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 7–16, IEEE Computer Society, 2004.

[52] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, "GRAPHITE : Polyhedral analyses and optimizations for GCC," in *Proceedings of the 2006 GCC Developers Summit*, p. 2006, Citeseer, 2006.

[53] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta, "Graphite two years after : First lessons learned from real-world polyhedral compilation," in *GCC Research Opportunities Workshop (GROW'10)*, 2010.

[54] D. Bernstein, M. Rodeh, and I. Gertner, "On the complexity of scheduling problems for parallel/pipelined machines," *IEEE Transactions on Computers*, vol. 38, pp. 1308–1313, Sept 1989.

[55] L. Lacassagne, D. Etiemble, A. Hassan-Zahraee, A. Dominguez, and P. Vezolle, "High level transforms for SIMD and low-level computer vision algorithms," in *Workshop on Programming Models for SIMD/Vector Processing (WPMVP, associated with ACM PPoPP)*, pp. 49–56, 2014.

[56] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, pp. 5–48, Mar. 1991.

[57] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, S. Torres, *et al.*, "Handbook of floating-point arithmetic," 2010.

[58] N. Revol and P. Théveny, "Numerical reproducibility and parallel computations : Issues for interval algorithms," *CoRR*, vol. abs/1312.3300, 2013.

[59] F. Jézéquel, P. Langlois, and N. Revol, "First steps towards more numerical reproducibility," *ESAIM : Proceedings*, vol. 45, pp. 229–238, Sept. 2014.

[60] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR : A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, June 2007.

[61] N. Revol and F. Rouillier, "Motivations for an arbitrary precision interval arithmetic and the MPFI library," *Reliable computing*, vol. 11, no. 4, pp. 275–290, 2005.

[62] W. Kahan, "Pracniques : Further remarks on reducing truncation errors," *Commun. ACM*, vol. 8, pp. 40–, Jan. 1965.

[63] A. Fog, *Instruction tables : Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2018. accessed version : 2018-04-27.

[64] P. Soderquist and M. Leeser, "Area and performance tradeoffs in floating-point divide and square-root implementations," *ACM Comput. Surv.*, vol. 28, pp. 518–564, Sept. 1996.

[65] C. Lomont, "Fast inverse square root," tech. rep., 2003.

[66] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, April 2018.

[67] B. Barrois, O. Sentieys, and D. Menard, "The hidden cost of functional approximation against careful data sizing : a case study," in *Proceedings of the Conference on*

Design, Automation & Test in Europe, pp. 181–186, European Design and Automation Association, 2017.

[68] P. Sebah and X. Gourdon, "Newton's method and high order iterations," tech. rep., 2001.

[69] V. Y. Pan, "Methods of computing values of polynomials," *Russian Mathematical Surveys*, vol. 21, no. 1, pp. 105–136, 1966.

[70] M. Bourgoin, E. Chailloux, and J. L. Lamotte, "SPOC : GPGPU programming through stream processing with OCaml," *Parallel Processing Letters*, vol. 22, p. 1240007, May 2012. 12 pages.

[71] JINJA2, "Python template engine." `http://jinja.pocoo.org/`.

[72] I. Masliah, M. Baboulin, and J. Falcou, "Metaprogramming dense linear algebra solvers applications to multi and many-core architectures," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 3, pp. 69–76, IEEE, 2015.

[73] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks : Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[74] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[75] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, May 2018.

[76] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading : Maximizing on-chip parallelism," in *Proceedings 22nd Annual International Symposium on Computer Architecture*, pp. 392–403, IEEE, June 1995.

[77] F. Jézéquel and J.-M. Chesneaux, "CADNA : a library for estimating round-off error propagation," *Computer Physics Communications*, vol. 178, no. 12, pp. 933–955, 2008.

[78] N. J. Higham, "Cholesky factorization," *Wiley Interdisciplinary Reviews : Computational Statistics*, vol. 1, no. 2, pp. 251–254, 2009.

[79] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3. JHU Press, 2012.

[80] K. Aneja, F. Laguzet, L. Lacassagne, and A. Merigot, "Video rate image segmentation by means of region splitting and merging," in *IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, 2009.

[81] L. Cabaret, L. Lacassagne, and D. Etiemble, "Parallel Light Speed Labeling : an efficient connected component labeling algorithm for multi-core processors," in *IEEE International Conference on Image Processing (ICIP)*, pp. 1–4, 2015.

[82] N. J. Higham, *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[83] J. J. Du Croz and N. J. Higham, "Stability of methods for matrix inversion," *IMA Journal of Numerical Analysis*, vol. 12, no. 1, pp. 1–19, 1992.

[84] SPIRAL, "Spiral : Sotfware/hardware generation for dsp algorithms." `http://www.spiral.net`.

[85] ATLAS, "Automatically tuned linear algebra software." `http://math-atlas.sourceforge.net`.

[86] T. Dong, A. Haidar, P. Luszczek, J. A. Harris, S. Tomov, and J. Dongarra, "LU factorization of small matrices : accelerating batched DGETRF on the GPU," in *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*, pp. 157–160, IEEE, 2014.

[87] T. Dong, A. Haidar, S. Tomov, and J. Dongarra, "A fast batched Cholesky factorization on a GPU," in *Parallel Processing (ICPP), 2014 43rd International Conference on*, pp. 432–440, IEEE, 2014.

[88] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[89] M. S. Grewal, "Kalman filtering," in *International Encyclopedia of Statistical Science*, pp. 705–708, Springer, 2011.

[90] E. Wolin and L. Ho, "Covariance matrices for track fitting with the kalman filter," *Nuclear Instruments and Methods in Physics Research Section A : Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 329, no. 3, pp. 493–500, 1993.

[91] A. Romero, M. Gouiffès, and L. Lacassagne, "Enhanced Local Binary Covariance Matrices (ELBCM) for texture analysis and object tracking," in *ACM International Conference on Computer Vision / Computer Graphics Collaboration Techniques and Applications*, 2013.

[92] D. Reid *et al.*, "An algorithm for tracking multiple targets," *IEEE transactions on Automatic Control*, vol. 24, no. 6, pp. 843–854, 1979.

[93] D. Beymer, P. McLauchlan, B. Coifman, and J. Malik, "A real-time computer vision system for measuring traffic parameters," in *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pp. 495–501, IEEE, 1997.

[94] R. G. Brown, P. Y. Hwang, *et al.*, *Introduction to random signals and applied Kalman filtering*, vol. 3. Wiley New York, 1992.

[95] A. Mohamed and K. Schwarz, "Adaptive kalman filtering for INS/GPS," *Journal of geodesy*, vol. 73, no. 4, pp. 193–203, 1999.

[96] J. Sasiadek and Q. Wang, "Sensor fusion based on fuzzy kalman filtering for autonomous robot vehicle," in *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, vol. 4, pp. 2970–2975, IEEE, 1999.

[97] A. C. Harvey, *Forecasting, structural time series models and the Kalman filter.* Cambridge university press, 1990.

[98] M. A. Palis and D. K. Krecker, "Parallel Kalman filtering on the connection machine," in *Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the*, pp. 55–58, IEEE, 1990.

[99] S. Li, D. C. Wunsch, E. O'Hair, and M. G. Giesselmann, "Extended Kalman filter training of neural networks on a SIMD parallel machine," *journal of Parallel and Distributed Computing*, vol. 62, no. 4, pp. 544–562, 2002.

[100] M.-Y. Huang, S.-C. Wei, B. Huang, and Y.-L. Chang, "Accelerating the Kalman filter on a GPU," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pp. 1016–1020, IEEE, 2011.

[101] T. Hu and M. Shing, "Computation of matrix chain products. Part I," *SIAM Journal on Computing*, vol. 11, no. 2, pp. 362–373, 1982.

[102] T. Hu and M. Shing, "Computation of matrix chain products. Part II," *SIAM Journal on Computing*, vol. 13, no. 2, pp. 228–251, 1984.

[103] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *ACM Sigplan Notices*, vol. 17, pp. 98–105, ACM, 1982.

[104] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, pp. 558–562, Nov. 1962.

[105] R. Frühwirth, "Application of Kalman filtering to track and vertex fitting," *Nuclear Instruments and Methods in Physics Research Section A : Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 262, no. 2, pp. 444–450, 1987.

[106] B. D. Anderson and J. B. Moore, *Optimal filtering*, vol. 21. 1979.

[107] H. E. Rauch, C. Striebel, and F. Tung, "Maximum likelihood estimates of linear dynamic systems," *AIAA journal*, vol. 3, no. 8, pp. 1445–1450, 1965.

[108] P. Billoir, "Track fitting with multiple scattering : A new method," *Nuclear Instruments and Methods in Physics Research*, vol. 225, no. 2, pp. 352 – 366, 1984.

[109] P. Kaminski, A. Bryson, and S. Schmidt, "Discrete square root filtering : A survey of current techniques," *IEEE Transactions on Automatic Control*, vol. 16, pp. 727–736, December 1971.

[110] D. H. Cámpora Pérez, "LHCb Kalman Filter cross architecture studies," *Journal of Physics : Conference Series*, vol. 898, no. 3, p. 032052, 2017.

[111] D. H. Cámpora Pérez and O. Awile, "An efficient low-rank kalman filter for modern simd architectures," *Concurrency and Computation : Practice and Experience*, p. e4483, 2018.

[112] G. Cerati, P. Elmer, S. Krutelyov, S. Lantz, M. Lefebvre, K. McDermott, D. Riley, M. Tadel, P. Wittich, F. Wurthwein, and A. Yagil, "Kalman filter tracking on parallel

architectures," *Journal of Physics : Conference Series*, vol. 898, no. 4, p. 042051, 2017.

[113] S. Gorbunov, U. Kebschull, I. Kisel, V. Lindenstruth, and W. Müller, "Fast SIMDized Kalman filter based track fit," *Computer Physics Communications*, vol. 178, no. 5, pp. 374–383, 2008.

[114] F. Lemaitre and L. Lacassagne, "Batched cholesky factorization for tiny matrices," in *Design and Architectures for Signal and Image Processing (DASIP), 2016 Conference on*, pp. 130–137, IEEE, 2016.

[115] F. Lemaitre, B. Couturier, and L. Lacassagne, "Cholesky factorization on SIMD multi-core architectures," *Journal of Systems Architecture*, vol. 79, pp. 1–15, Sept. 2017.

[116] F. Lemaitre, B. Couturier, and L. Lacassagne, "Small SIMD matrices for CERN high throughput computing," in *Workshop on Programming Models for SIMD/Vector Processing (WPMVP, associated with ACM PPoPP)*, p. 1, ACM, 2018.

Glossaire

Altivec Jeu d'instructions SIMD pour Power. largeur: 128 bits. Supporte uniquement les types 32 bits 15, 18, 20, 21, 44, 54, 129

ALU *Arithmetic and Logic Unit*: unité de calcul permettant de faire des opérations arithmétiques et logiques sur des entiers 17

AMD Fabricant de processeurs x86 16, 18, 19, 82

AoS *Array of Structures*: Agencement mémoire de type tableau de structures 23–26, 28, 32, 33, 55, 66, 67, 71, 72, 88, 94, 98, 101, 115

AoSoA *Array of Structures of Arrays*: Agencement mémoire hybride 30–33, 55, 66, 67, 69, 71, 72, 83, 84, 86, 88, 94, 98, 101, 102, 114, 115

ARM Architecture matérielle spécifiée par la société éponyme 15, 19–21, 62, 81, 82, 128, 129

Assembleur Langage de programmation bas niveau. Permet une traduction immédiate depuis/vers le langage machine binaire 16, 20, 86, 127

AVX Jeu d'instructions SIMD pour x86. largeur: 256 bits 15, 19, 20, 42, 44, 67, 83, 113

AVX512 Jeu d'instructions SIMD pour x86. largeur: 512 bits 15, 20, 41–44, 82, 113

Batch Données regroupés afin d'être traitées ensemble 62, 63, 65–72, 77, 84, 85, 88, 94, 115

Boost.SIMD Bibliothèque C++ pour l'écriture de code SIMD 16, 87, 88

C Langage de programmation bas niveau. Considéré comme un langage assembleur portable 16, 17, 32, 33, 39, 49–51, 128

C++ Langage de programmation bas niveau avec des fonctionnalités haut niveau 10, 49, 51, 127–129

Eigen Bibliothèque d'algèbre linéaire pour CPU 12, 82–84, 115

Exactitude Proximité de la valeur calculée à la valeur réelle (aussi appelé "précision de calcul", ou *accuracy* en anglais 39–41, 45, 49, 55, 74

FMA *Fused Multiply-Add*: Multiplie puis additionne en effectuant un seul arrondi 19, 43, 45–49, 82

Fortran Langage de programmation orienté calculs scientifiques 33

FPU *Floating Point Unit*: unité de calcul pour les nombres à virgule flottante 17

gcc Compilateur C et C++ produit par le projet GNU 35, 81

IBM 15, 20, 82

icc Compilateur C et C++ d'Intel 73, 81

IEEE 754 Norme pour la représentation de nombres flottants 19, 39, 40, 44, 55, 81

Intel Fabricant de processeurs x86 16, 19, 20, 60, 82, 128, 129

ISA *Instruction Set Architecture*: Jeu d'instruction 44, 82

ISPC langage de programmation parallèle proche du C 16

Jinja2 Moteur de template 49–53, 55, 115

Libsimdpp Bibliothèque C++ pour l'écriture de code SIMD 16, 87

Linux Système d'exploitation 60

Magma Bibliothèque d'algèbre linéaire pour CPU/GPU 12, 115

MIPP Bibliothèque C++ pour l'écriture de code SIMD 16, 87

MKL Bibliothèque d'algèbre linéaire pour x86 par Intel 12, 82–84, 88, 94, 115, 116

MPFR Bibliothèque de calcul flottant en précision arbitraire 40, 41, 73

Neon Jeu d'instructions SIMD pour ARM. largeur: 128 bits 15, 20, 21, 43, 44, 47, 82

OpenACC Extension du C permettant la programmation parallèle (SIMD ou multi-threading) en utilisant des `#pragma`s 16

OpenCL Extension du C permettant la programmation parallèle hétérogène 16, 49

OpenMP Extension du C permettant la programmation parallèle (SIMD ou multi-threading) en utilisant des `#pragma`s 16, 70–72

PE *Processing Element*: petites unités de calcul indépendantes coordonnées par un processeur unique 17

PHP Langage de programmation et de balisage. Permet de générer facilement du HTML 49

Power Architecture matérielle spécifiée par la société IBM 18, 21, 62, 81, 127, 129

Précision Quantité d'information utilisée pour représenter la valeur (aussi appelé "précision de données", ou *precision* en anglais) 39–41, 55, 73, 74

Python Langage de programmation 49, 50, 53, 55, 115

RAM *Random Access Memory*: mémoire à accès aléatoires 22, 67, 82, 113

SIMD *Single Instruction Multiple Data*: une même opération s'appliquant en parallèle sur plusieurs données différentes 12, 15–25, 28, 32, 33, 40, 43, 49, 53–55, 66–69, 71, 72, 82–84, 86, 87, 90, 94, 95, 98, 101, 103, 113–116, 127–129

SMT *Simultaneous Multithreading*: Technique consistant à exécuter plusieurs *threads* en même temps sur un même cœur, se partageant ainsi les unités fonctionelles 71

SoA *Structure of Arrays*: Agencement mémoire de type structure de tableaux 25, 27, 28, 30, 32, 33, 52, 55, 66, 67, 69, 71, 72, 98, 115

SSE Jeu d'instructions SIMD pour x86. largeur: 128 bits 15, 19, 21, 28, 42, 44, 54, 83, 113

SVE Jeu d'instructions SIMD pour ARM. largeur: entre 128 et 2048 bits 20, 21

SWAR *SIMD Within A Register*: Architecture bénéficiant d'instructions utilisant un registre comme un vecteur de plusieurs éléments plus petits, et traitant ces éléments en parallèle 18–20

Thread Fil d'exécution 32, 33, 61, 71, 72

UME::SIMD Bibliothèque C++ pour l'écriture de code SIMD 16, 87

Vc Bibliothèque C++ pour l'écriture de code SIMD 16

Vcl Bibliothèque C++ pour l'écriture de code SIMD 16, 87

VSX Jeu d'instructions SIMD pour Power. largeur: 128 bits. Extension du jeu d'instruction Altivec 15, 20, 21, 43, 44, 82

x86 Architecture matérielle initiée par Intel 15, 16, 18, 21, 60, 61, 81, 127–129