

Abstraction des jeux d'instructions vectoriels, application sur l'architecture RISC-V et sur des solveurs HPC

Rapport de stage

2026-08-19

Auteur

MULOT-RADOJCIC Ivan

Encadrants

CASSAGNE Adrien, GAYNO Raphaël, POPOV Mihail

Remerciements

Je préface ce rapport en remerciant chaleureusement **Adrien Cassagne**, **Raphaël Gayno** et **Mihail Popov** de m'avoir retenu comme candidat pour ce stage.

Je remercie également **Karim Mohamed El Maarouf**, **Ani Anciaux** et **Denis Barthou**, qui ont participé aux réunions hebdomadaires du stage. Leurs expertises, conseils et suggestions ont été des facteurs clefs dans l'avancement de ce travail.

L'intérêt porté à MIPP par l'ensemble des personnes avec qui j'ai pu échanger et collaborer durant ce stage a été une grande source de motivation.

Je remercie également **Karim Mohamed El Maarouf**, **Ani Anciaux** et **Raphaël Gayno** pour leur accueil au sein de l'IFPEN.

Je tiens enfin à remercier plus particulièrement **Adrien Cassagne**, qui m'a accompagné quotidiennement tout au long du stage. Son implication, ses conseils et la confiance qu'il m'a accordée ont été des moteurs importants dans la réalisation de ce travail.

Merci également à l'ensemble de l'équipe ALSOC, doctorants, permanents et stagiaires, pour leur accueil durant ces six mois.

MIPP est le fruit d'un travail collectif, et je suis fier de contribuer à ce projet.

Table des matières

1	Introduction	4
2	Contexte	5
2.1	Les extensions SIMD	5
2.2	L'extension SIMD de RISC-V (RVV)	6
2.3	Les ordinateurs monocartes RISC-V considérés	9
2.3.1	La Banana Pi BPI-F3	9
2.3.2	Le Sipeed K3 Pico-ITX	11
2.4	Les plateformes x86 considérées	12
2.5	État de l'Art	12
2.5.1	L'intégration du SIMD dans les compilateurs	13
2.5.2	Programmation avec des fonctions intrinsèques	14
2.5.3	Les bibliothèques d'abstraction SIMD	14
2.6	La bibliothèque MIPP	15
2.6.1	Positionnement de MIPPv2	19
3	Travaux effectués	19
3.1	Changements structurels au générateur d'en-têtes MIPP	19
3.2	Implémentation de RVV dans MIPP	22
3.3	Enrichissement de MIPP	23
3.3.1	Évolutions fonctionnelles et architecturales	23
3.3.2	Générateur de tests	25
3.3.3	Générateur de documentation	26
4	Mesures de performances & intégration à des applications HPC	27
4.0.1	Benchmarks d'algèbre linéaire	28
4.0.2	Benchmarks de lois physiques	33
4.0.3	Benchmark micro-kernels GEMM	36
4.0.4	Conclusion des mesures de performance	45
5	Conclusion, Perspectives & Ouvertures	46
5.1	Conclusion	46
5.2	Perspectives	47
5.3	Ouvertures	48
6	Annexes	48
	Bibliographie	67

1 | Introduction

Dans le domaine du calcul haute performance (*High-Performance Computing*, HPC), l'exploitation des extensions vectorielles et du paradigme *Single Instruction Multiple Data* (SIMD) est indispensable pour tirer pleinement parti des processeurs modernes (SSE, AVX2, AVX-512, NEON, SVE, RVV). Cependant, exploiter efficacement ces unités soulève plusieurs difficultés. L'auto-vectorisation intégrée aux compilateurs offre peu de contrôle et de garanties de performance, tandis que la programmation directe par fonctions intrinsèques, bien que performante, nuit à la portabilité et complexifie la maintenance du code.

La bibliothèque *open-source* MIPP (**My Intrinsics Plus Plus**) [1] propose une interface unifiée et portable à surcoût nul (*zero-cost abstraction*). Ce projet académique, initialement développé par **Adrien Cassagne**, est aujourd'hui utilisé dans plusieurs projets académiques et industriels. Alors que la première version était directement écrite en C++, l'émergence de nouvelles extensions SIMD comme l'extension vectorielle V de RISC-V (RVV) ou ARM SVE a motivé la création d'une nouvelle version MIPPv2. Cette dernière repose sur un générateur Python produisant des en-têtes C++, afin de faciliter la maintenance et l'évolution de la bibliothèque. Dans ce rapport, sauf indication contraire, le terme « MIPP » désignera MIPPv2.

L'intégration de l'extension vectorielle RISC-V Vector 1.0 (RVV) a nécessité une refonte structurelle de ce générateur. RVV introduit en effet des mécanismes spécifiques tels que le groupement dynamique de registres (*Length Multiplier*, LMUL) ou la division vectorielle (LDIV). Dans ce cadre, les principales contributions présentées dans ce rapport sont les suivantes :

- **support complet de RVV 1.0 et refonte du générateur** : intégration native de l'ensemble des types et opérations RVV, modélisation générique de LMUL et LDIV (supportés nativement en RVV, émulés pour les autres jeux d'instructions ou ISA, *Instruction Set Architecture*), formalisation et unification des motifs de masquage (*masks*, *mask*, *maskz*), support simultané de plusieurs ISA au sein d'un même programme, et création d'une ISA scalaire paramétrable servant d'oracle sémantique et de repli automatique (*scalar fallback*).
- **infrastructure de test, assurance qualité et documentation** : développement d'un générateur automatisé de tests unitaires (Catch2) comparant les résultats à l'oracle scalaire avec prise en compte des tolérances numériques et des dépassements d'entiers, mise en place d'une intégration continue (*Continuous Integration*, CI) multi-architectures sur le cluster *Dalek* (LIP6), et prototype de générateur de documentation (MkDocs).
- **validation expérimentale et intégration industrielle** : validation de la compatibilité de MIPPv2 au sein du solveur linéaire MCGSolver de l'IFPEN via l'en-tête MIPPv2.h, campagne d'évaluation de performance sur trois microarchitectures (Intel

Skylake, AMD Zen4, SpacemiT K3 X100), et caractérisation empirique des situations où la vectorisation explicite surpasse l'auto-vectorisation du compilateur.

Ce rapport s'articule comme suit : la Section 2 présente les principes de la programmation vectorielle, le fonctionnement de RVV et de MIPP, les plateformes matérielles d'évaluation ainsi que l'état de l'art des solutions de programmation SIMD. La Section 3 détaille les développements apportés au générateur et à l'infrastructure logicielle. La Section 4 analyse les résultats de performance obtenus sur les différentes architectures et l'intégration dans MCGSolver. Enfin, la Section 5 dresse le bilan de ces travaux et esquisse les perspectives d'évolution de la bibliothèque.

2 | Contexte

Cette section pose le cadre théorique, matériel et logiciel indispensable à l'analyse des travaux de ce rapport. Sont d'abord introduits les principes du calcul vectoriel et les spécificités de l'extension RVV (VLA, LMUL, prédication). Sont ensuite présentées les plateformes matérielles d'évaluation, composées d'ordinateurs monocartes RISC-V (SpacemiT K1, SpacemiT K3) et de processeurs x86 (Skylake, Zen4). La section examine ensuite l'état de l'art des solutions de vectorisation, de l'auto-vectorisation des compilateurs aux bibliothèques d'abstraction telles que *Google Highway* et VCL, avant de détailler l'architecture logicielle, le générateur et le positionnement de la bibliothèque MIPP.

2.1 | Les extensions SIMD

Dans la taxonomie de Flynn [2], le modèle *Single Instruction Multiple Data* (SIMD) désigne une classe d'architectures où une unique instruction s'applique simultanément à plusieurs éléments de données indépendants. Bien que ces deux concepts diffèrent historiquement, le terme « extension vectorielle » est couramment employé pour désigner les jeux d'instructions SIMD. L'un des exemples les plus répandus est l'extension AVX (*Advanced Vector Extensions*) pour le jeu d'instructions x86.

Contrairement au modèle scalaire classique *Single Instruction Single Data* (SISD), une instruction SIMD opère sur des registres vectoriels regroupant plusieurs données. Ce parallélisme au niveau des données (*Data-Level Parallelism*, DLP) permet d'accroître significativement le débit de calcul d'une application. Sur le plan matériel, la latence et le débit (en cycles processeur) d'une instruction vectorielle sont en effet comparables à ceux de l'instruction scalaire équivalente, ce qui permet de traiter plusieurs valeurs dans un intervalle de temps similaire (Fig. 1).

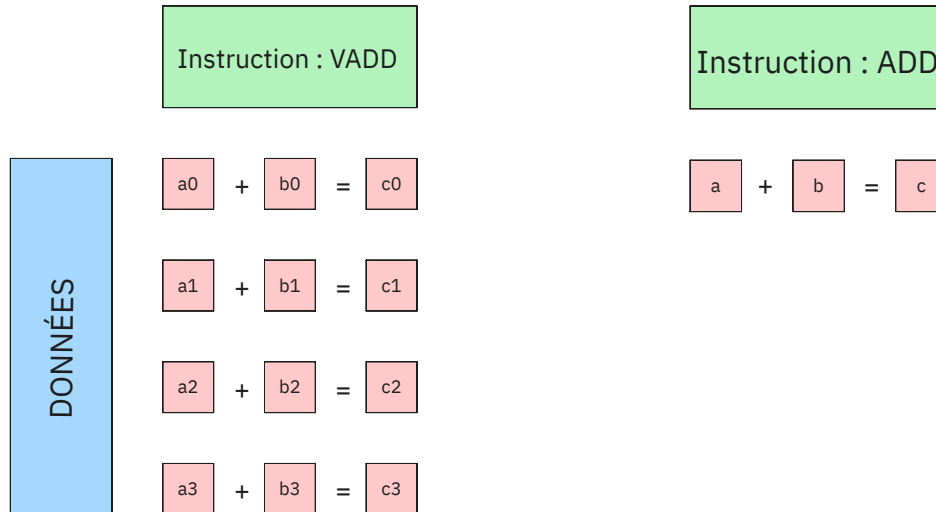


Fig. 1 – Comparaison d’une addition sur des registres vectoriels contenant quatre données et sur des registres scalaires.

Aujourd’hui, des extensions **SIMD** sont intégrées à la quasi-totalité des architectures généralistes : **SSE** [3], **AVX**, **AVX2** et **AVX-512** [4] pour **x86** ; **NEON** [5] et **SVE** [6] pour **ARM** ; et, depuis 2021, **RVV 1.0** [7] pour **RISC-V**.

2.2 | L’extension **SIMD** de **RISC-V** (**RVV**)

RVV est l’extension **SIMD** de l’architecture **RISC-V**. Elle se distingue de la plupart des extensions **SIMD** traditionnelles par sa conception *Vector Length Agnostic* (**VLA**), une approche également adoptée par l’extension **SIMD ARM SVE**.

Une extension **SIMD** est dite **VLA** lorsque le jeu d’instructions ne fige pas la largeur matérielle des registres vectoriels. Les binaires compilés peuvent ainsi s’exécuter sur n’importe quelle implémentation matérielle compatible sans modification ni recompilation.

La spécification de **RVV** [7] définit 32 registres vectoriels notés **v0** à **v31**. Elle définit le paramètre **ELEN** (*Element Length*) comme la taille maximale (en bits) d’un élément de vecteur ($\text{ELEN} \geq 8$ bits, devant être une puissance de deux). Elle impose également que la taille physique d’un registre vectoriel, notée **VLEN** (*Vector Length in bits*), respecte la contrainte :

$$\text{ELEN} \leq \text{VLEN} \leq 2^{16} \text{ bits}$$

Ainsi, un processeur **RVV** doté de registres de 128 bits ($\text{VLEN} = 128$) et un processeur équipé de registres de 1024 bits ($\text{VLEN} = 1024$) exécutent le même code binaire vectorisé sans nécessiter d’adaptation (Fig. 2).

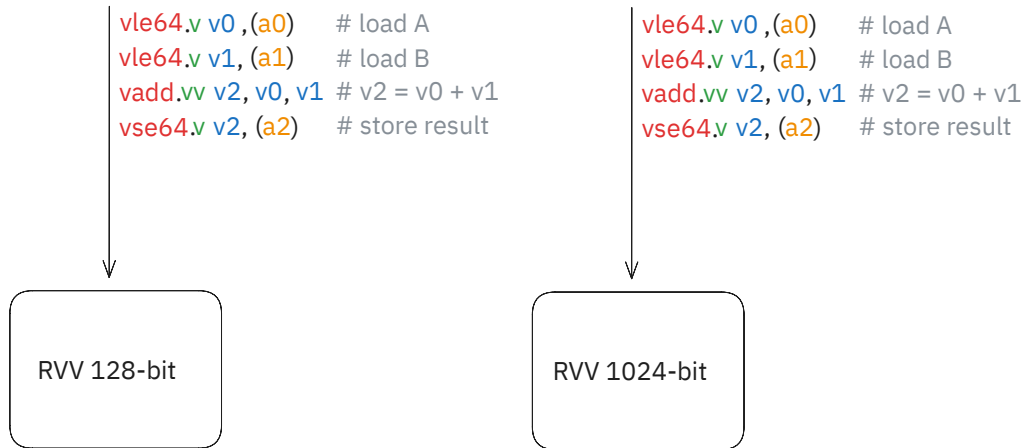


Fig. 2 – Illustration du modèle VLA en RVV : un même code binaire s'exécute sur des registres de taille variable (128 bits à gauche, 1024 bits à droite).

RVV permet également de reconfigurer dynamiquement la longueur de traitement et la taille des vecteurs manipulés en cours d'exécution. Pour cela, l'architecture introduit des registres de contrôle et d'état (*Control and Status Registers*, CSR) dédiés : *vtype* (*Vector Type*), qui stocke la configuration vectorielle active, et *vl* (*Vector Length*), qui contient le nombre effectif d'éléments traités par les instructions (Fig. 3).

Le registre *vtype* définit notamment deux paramètres fondamentaux :

- *vsew* (*Selected Element Width*), la largeur en bits d'un élément de donnée (8, 16, 32 ou 64 bits).
- *vlmul* (*Vector Register Group Multiplier*), le facteur multiplicateur $\in \{\frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1, 2, 4, 8\}$ permettant de regrouper virtuellement plusieurs registres physiques pour former un registre vectoriel logique plus large (ou à l'inverse de n'en utiliser qu'une fraction).

La configuration de ces registres s'effectue au moyen d'instructions dédiées telles que *vsetvli*, *vsetivli* ou *vsetvl* [8].

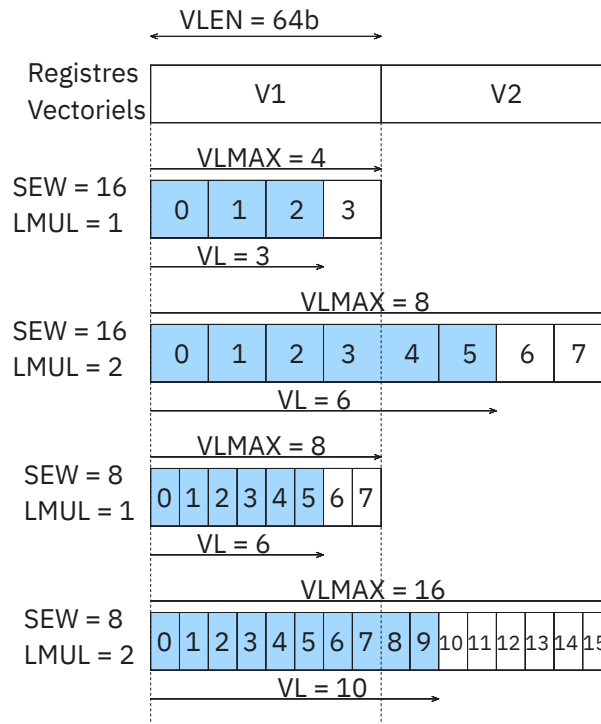


Fig. 3 – Illustration de la configuration dynamique de la taille des vecteurs en RVV.

RVV supporte également l'exécution **prédiquée** (ou masquée) : la majorité des instructions vectorielles peuvent recevoir un registre de masque (généralement `v0`) en paramètre. Ce masque booléen spécifie les éléments du vecteur résultat devant être mis à jour ou conservés (Fig. 4).

`vadd.vv v4, v1, v2, vm` # $v4[i] = vm[i] ? v1[i] + v2[i] : v4[i]$

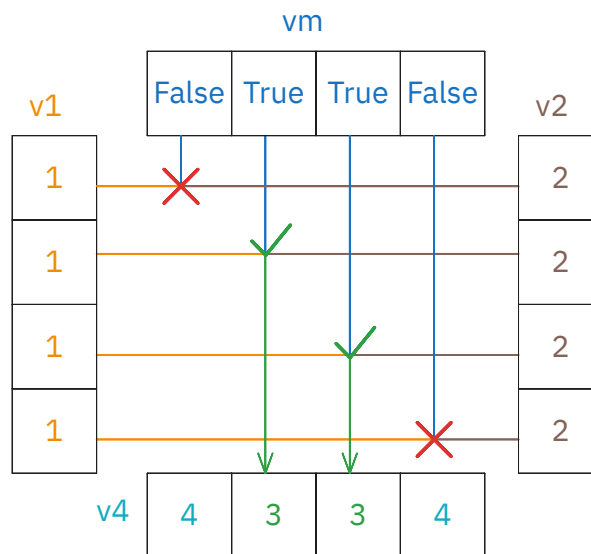


Fig. 4 – Illustration d'un `vadd` masqué, en supposant que chaque élément de `v4` vaut initialement 4 et que les vecteurs font quatre éléments.

Afin d'exploiter les instructions RVV en C/C++, une interface standardisée est définie dans l'en-tête `riscv_vector.h` [9]. Cet en-tête expose les types opaques correspondant aux registres vectoriels ainsi que l'ensemble des fonctions intrinsèques associées.

Les types de données définis dans `riscv_vector.h` encodent explicitement la largeur effective de l'élément (*Effective Element Width*, *EEW*) ainsi que le facteur de groupement actif (*Effective Multiplier*, *EMUL*). Les types de masques booléens (`vboolN_t`) dépendent quant à eux du ratio $\frac{EEW}{EMUL}$. L'inventaire complet de ces types est détaillé en annexe (Tableau 2, Tableau 3, Tableau 4).

Les fonctions intrinsèques de `riscv_vector.h` adoptent la nomenclature normalisée suivante :

```
__riscv_{instruction}_{opérandes}_{type de retour}_{mode d'arrondi}_{politique}
```

- `instruction` désigne l'opération vectorielle ciblée (`vadd`, `vse64`, `vfmadd`, etc.).
- `opérandes` indique la nature des opérandes (`v` pour vecteur, `vx` pour vecteur-scalaire, `vf` pour vecteur-flottant, etc.).
- `type de retour` précise le type et le facteur *LMUL* du résultat (`i32m1`, `i64mf2`, `f32m2`, etc.).
- `mode d'arrondi` (optionnel) spécifie le comportement d'arrondi pour les calculs en virgule flottante ou point fixe.
- `politique` (optionnel) définit la politique de préservation des éléments inactifs et hors-limite (`ta/tu` pour *tail-agnostic / tail-undisturbed*, `ma/mu` pour *mask-agnostic / mask-undisturbed*).

Notons que la spécification RVV propose également une interface surchargée (*overloaded intrinsics*) déduisant automatiquement les types. Cependant, le générateur de MIPP cible la forme complète et non surchargée.

2.3 | Les ordinateurs monocartes RISC-V considérés

Pour le développement, l'intégration continue et l'évaluation expérimentale de MIPP, deux ordinateurs monocartes (*Single Board Computer*, *SBC*) architecturés autour du jeu d'instructions RISC-V sont utilisés : la **Banana Pi BPI-F3** [10] (BPI-F3) et la **Sipeed K3 Pico-ITX** [11] (SIP-K3). Ces deux cartes sont accessibles depuis le *cluster Dalek* [12] de l'équipe ALSOC du LIP6. En effet, cela a permis de développer directement sur une plateforme RISC-V, facilitant ainsi le déploiement, les tests et le débogage du code.

2.3.1 | La Banana Pi BPI-F3

La BPI-F3 (Fig. 5) est un SBC équipé d'un processeur **SpacemiT K1** [13]. SpacemiT [14] est une entreprise basée à Hangzhou, en Chine, spécialisée dans la conception de processeurs RISC-V destinés à l'intelligence artificielle (IA) et au HPC.

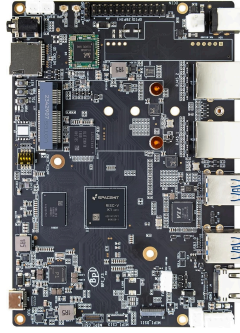


Fig. 5 – Photographie de la BPI-F3.

Le **SpacemiT K1** est un processeur RISC-V à huit cœurs prenant en charge RVV 1.0, dont la longueur des registres vectoriels RVV (VLEN) est de 256 bits. Les cœurs sont des modèles **X60** cadencés à 1,6 GHz. Chacun dispose d'un cache L1d et d'un cache L1i de 32 Kio. Les cœurs sont regroupés en deux *clusters* de quatre cœurs. Chaque cluster est doté d'un cache L2 partagé de 512 Kio (soit 1 Mio de cache L2 au total sur la puce). La Fig. 6 extraite du site de Banana Pi [15] illustre la topologie de ce *System on Chip* (SoC).

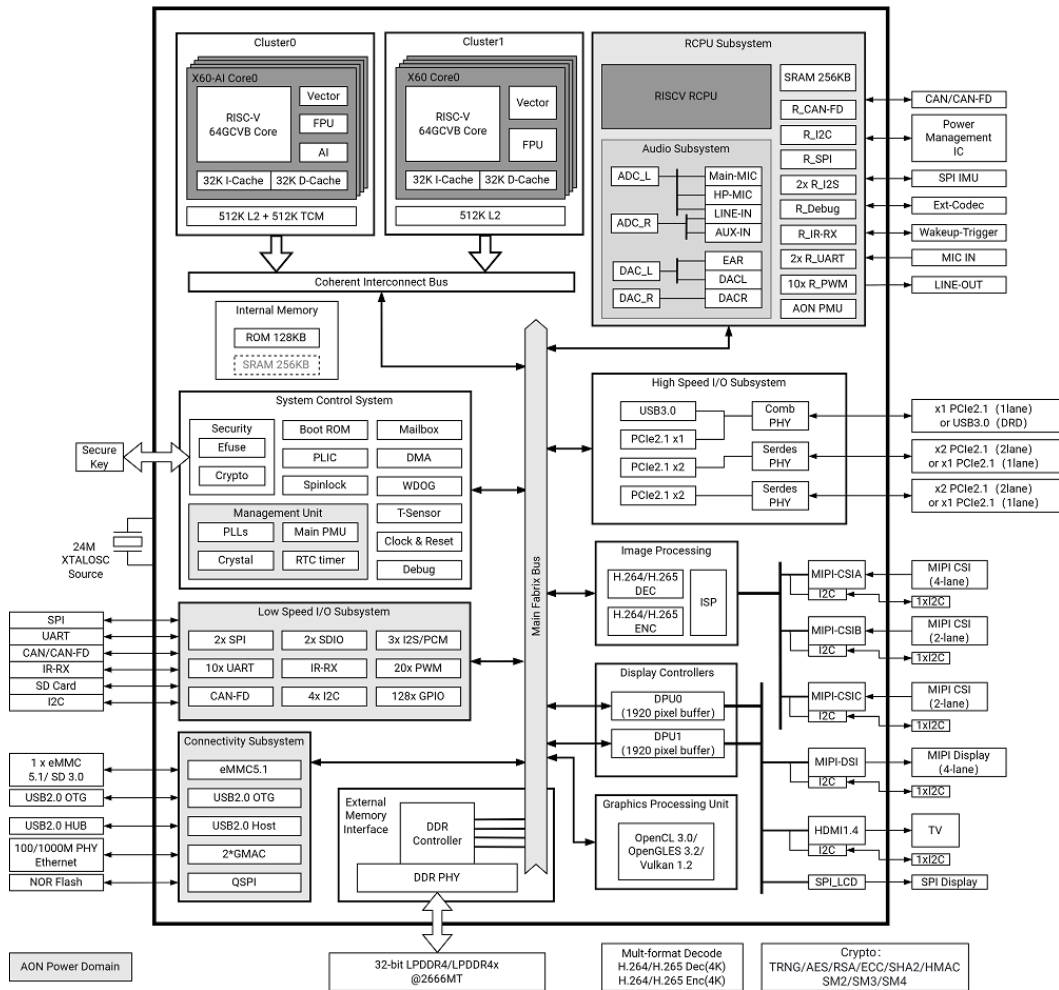


Fig. 6 – Topologie du SpacemiT K1 [15].

2.3.2 | Le Sipeed K3 Pico-ITX

La SIP-K3 (Fig. 7) est un SBC équipé d'un processeur **SpacemiT K3** [16].

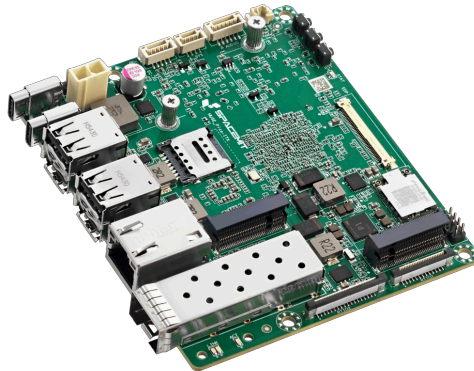


Fig. 7 – Photographie de la carte SIP-K3.

Le **SpacemiT K3** constitue la nouvelle génération de processeurs RISC-V développée par SpacemiT. Il repose sur une architecture hétérogène composée de huit cœurs X100 et de huit cœurs A100. Les cœurs X100 représentent la nouvelle génération de cœurs orientés HPC de SpacemiT et succèdent directement aux cœurs X60. Ils sont cadencés à 2,4 GHz, prennent en charge RVV 1.0 et disposent de registres vectoriels d'une longueur (VLEN) de 256 bits. Chaque cœur X100 intègre un cache L1d et un cache L1i de 64 Kio. Les huit cœurs X100 sont répartis en deux *clusters* de quatre cœurs, chacun doté d'un cache L2 partagé de 4 Mio (soit 8 Mio de cache L2 au total pour la partie HPC). Les cœurs A100 sont des cœurs RISC-V conçus pour l'exécution d'applications d'IA. Ils intègrent des extensions spécifiques développées par SpacemiT pour accélérer les charges de calcul liées à l'IA, notamment les opérations d'inférence. Leur microarchitecture diffère de celle des cœurs X100. Ils disposent de caches L1d et L1i de 32 Kio, prennent en charge RVV 1.0 avec des registres vectoriels étendus à 1024 bits et sont cadencés à 2,0 GHz. Les huit cœurs A100 sont organisés en deux *clusters* de quatre cœurs, chacun associant 1 Mio de cache L2 et 1,5 Mio de mémoire TCM (*Tightly-Coupled Memory*).

La Fig. 8 issue du site de SpacemiT [17] illustre la topologie de ce SoC.

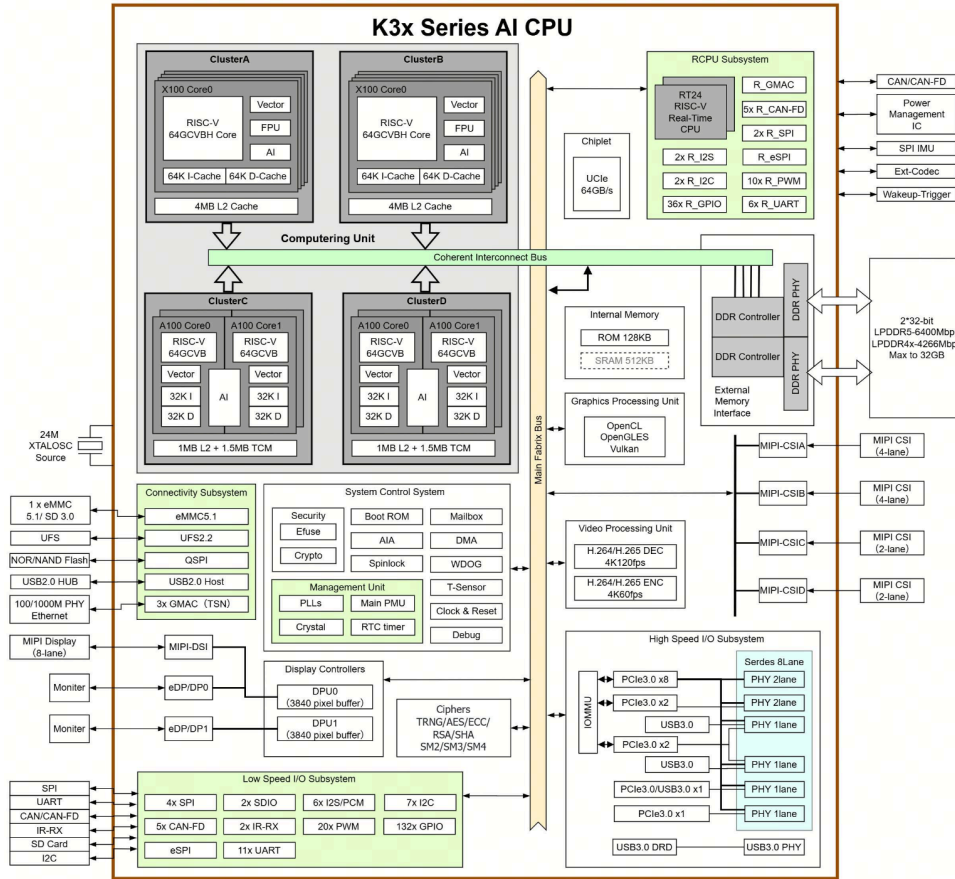


Fig. 8 – Topologie du SpacemiT K3 [17].

Les cœurs A100 peuvent être considérés comme des accélérateurs dédiés. Il est toutefois important de souligner qu'ils prennent en charge l'ensemble des extensions RISC-V supportées par les cœurs X100, à l'exception de l'extension Hyperviseur (H).

2.4 | Les plateformes x86 considérées

En complément des cibles RISC-V, deux plateformes x86 sont utilisées pour les évaluations de performance : Zen4 (AMD Ryzen 9 7945HX) : microarchitecture moderne (2022) prenant en charge l'ensemble des extensions SIMD SSE, AVX, AVX2 et AVX-512 et Skylake (Intel Core i5-6200U) : microarchitecture plus ancienne (2015) restreinte aux extensions SSE, AVX et AVX2.

2.5 | État de l'Art

La programmation vectorielle sur architectures modernes se décline selon trois approches principales, offrant des compromis distincts entre niveau d'abstraction, portabilité du code source et maîtrise de la performance. Sont d'abord abordées les solutions intégrées aux compilateurs et aux standards de langage (auto-vectorisation, extensions dédiées et proposition `std::simd`). Sont ensuite examinées les approches à bas niveau reposant sur l'usage direct de fonctions intrinsèques. Enfin, un panorama

des bibliothèques d'abstraction SIMD existantes (*Google Highway*, *VCL*, *SLEEP*) permet de situer les choix de conception de MIPP.

2.5.1 | L'intégration du SIMD dans les compilateurs

L'exploitation des capacités vectorielles peut être directement confiée à la chaîne de compilation ou au langage de programmation, sans recourir à des bibliothèques tierces.

L'auto-vectorisation est une phase d'optimisation intégrée aux compilateurs C/C++ modernes (GCC, Clang, ICC, MSVC). Elle consiste en la transformation automatique d'un code écrit de manière scalaire en séquences d'instructions vectorielles. Pour que le compilateur cible une extension SIMD précise, des drapeaux d'optimisation doivent lui être fournis (par exemple `-mavx2` pour AVX2, `-mavx512f` pour AVX-512, ou l'option générique `-march=native` qui adapte la génération de code au processeur hôte).

L'application de l'auto-vectorisation n'est toutefois pas systématique et requiert la validation de deux critères par le compilateur :

- la conformité sémantique : le code vectorisé généré doit produire un résultat strictement équivalent au code scalaire (absence de dépendances de données entravant la parallélisation et absence d'alias mémoire potentiel entre pointeurs).
- la rentabilité estimée : le modèle de coût (*cost model*) du compilateur doit prédire que le gain de débit vectoriel l'emporte sur les pénalités éventuelles (alignement mémoire, gestion des résidus de boucle).

Le programmeur délègue ainsi l'ensemble des choix vectoriels au compilateur. Si cette approche garantit la portabilité du code source, elle offre peu de contrôle sur le code assembleur final et s'avère fragile dès que les motifs d'accès mémoire ou les flux de contrôle se complexifient.

Certains compilateurs, comme GCC ou Clang, proposent également des extensions de langage dédiées [18]. Ces *GNU Vector Extensions* permettent d'exprimer explicitement des opérations vectorielles en définissant des types à l'aide de l'attribut `vector_size` :

```
typedef int v4si __attribute__((vector_size (16)));
```



Contrairement aux intrinsèques, ces extensions ne garantissent pas l'utilisation d'une instruction SIMD spécifique, mais décrivent un vecteur de données que le compilateur traduit selon l'architecture cible. Ces extensions ne sont toutefois pas standardisées et demeurent non portables vers d'autres compilateurs comme MSVC.

Afin de normaliser la programmation vectorielle au sein du langage, la proposition `std::simd` [19] vise à introduire une couche d'abstraction portable dans la bibliothèque standard C++ (destinée au standard C++26). Elle permet de manipuler des vecteurs de données scalaires via des types dédiés et des surcharges d'opérateurs arithmétiques, masquant ainsi l'extension SIMD sous-jacente. La proposition définit également le type

de masque `std::simd_mask` pour gérer les opérations conditionnelles prédiquées, ainsi que des surcharges vectorielles pour les fonctions mathématiques élémentaires (`exp`, `log`, `cos`).

2.5.2 | Programmation avec des fonctions intrinsèques

À l’opposé de l’auto-vectorisation, la programmation par fonctions intrinsèques (*intrinsics*) confère au développeur un contrôle explicite sur la vectorisation de son code. Pour chaque extension SIMD (SSE, AVX, NEON, RVV), il existe un fichier d’en-tête dédié (`immintrin.h`, `arm_neon.h`, `riscv_vector.h`). Ces en-têtes définissent des types de données permettant de manipuler directement des variables associées aux registres vectoriels (par exemple `__m128`, `float32x4_t` ou `vfloat32m1_t` pour des registres contenant des flottants 32 bits). Ils fournissent également un ensemble de fonctions appelées *fonctions intrinsèques*. Sauf pour quelques rares exceptions, chaque fonction intrinsèque correspond directement à une instruction assembleur de l’extension SIMD ciblée.

Cette approche offre un contrôle fin sur les instructions générées, mais présente deux limites majeures :

- une syntaxe verbeuse et fortement couplée au matériel, dépourvue de généricité de types et de surcharges d’opérateurs arithmétiques.
- une absence de portabilité : un code écrit avec les intrinsèques d’une ISA (par exemple SSE) ne peut pas être compilé pour une autre architecture telle qu’ARM ou RISC-V, contraignant le développeur à maintenir plusieurs implémentations distinctes d’un même algorithme.

C’est pour concilier contrôle de la vectorisation et portabilité du code source que des bibliothèques d’abstraction SIMD ont vu le jour.

2.5.3 | Les bibliothèques d’abstraction SIMD

Une bibliothèque d’abstraction SIMD fournit une interface unifiée au-dessus des instructions vectorielles matérielles, permettant d’exprimer des calculs vectoriels portables sans sacrifier l’efficacité à l’exécution.

L’une des bibliothèques de référence dans ce domaine est *Highway* [20], développée par Google. *Highway* prend en charge un ensemble exhaustif d’architectures modernes : x86 (SSE, AVX, AVX-512), ARM (NEON, SVE), RISC-V (RVV), PowerPC, IBM Z, WebAssembly SIMD et LoongArch. Sa conception vise la portabilité de performance (*performance portability*) et supporte nativement les extensions SIMD à taille de registres variable (VLA) comme RVV ou SVE.

La fonctionnalité phare d’*Highway* est l’aiguillage dynamique (*dynamic dispatch*). Grâce à un mécanisme d’espaces de nommage (*namespace*) et à la macro `HWY_EXPORT`, la bibliothèque compile automatiquement plusieurs variantes vectorielles d’un même *kernel* au sein d’un binaire unique. À l’exécution, lors du premier appel, un pointeur de fonction est initialisé vers l’implémentation la plus performante supportée par

la machine hôte (par exemple AVX-512, puis AVX2, puis SSE sur x86, avec un repli automatique vers l'implémentation scalaire si nécessaire). Les appels suivants utilisent directement cette implémentation sans surcoût d'indirection. *Highway* constitue ainsi une bibliothèque mature et puissante, disposant d'un riche ensemble de primitives SIMD facilitant l'écriture de *kernels* vectorisés portables.

Une autre solution notable est la bibliothèque *Vector Class Library* (VCL [21]), développée par Agner Fog. VCL est une bibliothèque *header-only* ciblant exclusivement les extensions SIMD de l'architecture x86. Elle définit des types vectoriels fortement typés ainsi que des surcharges d'opérateurs C++, offrant une syntaxe expressive et un contrôle direct des registres x86. Cependant, dans VCL, la largeur des registres est explicitement encodée dans le type (*Vec4f* pour 128 bits, *Vec8f* pour 256 bits, *Vec16f* pour 512 bits), ce qui lie le code source à une taille de registre donnée.

La première version de MIPP (MIPPv1 [1]) partage avec VCL l'usage de la métaprogrammation par *templates* pour abstraire les intrinsèques matériels, mais s'en distingue par son modèle de portabilité : plutôt que de figer la largeur dans le type, MIPPv1 introduit un type générique *Reg<T>* dont la taille matérielle s'adapte statiquement à l'extension SIMD activée à la compilation (SSE, AVX2, AVX-512 ou ARM NEON). Un même algorithme rédigé avec *Reg<float>* opère ainsi de manière transparente sur 4, 8 ou 16 éléments selon la cible, assurant la portabilité du code source sans surcoût à l'exécution.

D'autres bibliothèques généralistes notables dans l'écosystème C++ sont EVE (*Expressive Vector Engine*), *nsimd*, *Vc* ou *simdpp*.

Enfin, la plupart des bibliothèques SIMD généralistes se limitent aux opérations arithmétiques de base et n'intègrent pas les fonctions mathématiques élémentaires (*sin*, *cos*, *exp*, *log*). La vectorisation de ces fonctions exige des algorithmes spécifiques, souvent basés sur des approximations polynomiales, des réductions d'intervalle et un contrôle rigoureux de l'erreur numérique. Des bibliothèques spécialisées comme SLEEF (*SIMD Library for Evaluating Elementary Functions* [22]) ont donc été développées pour combler ce manque en fournissant des implémentations vectorisées, hautement optimisées et portables de ces fonctions mathématiques.

2.6 | La bibliothèque MIPP

My Intrinsics Plus Plus (MIPP [1]) est une bibliothèque *header-only* conçue pour unifier et simplifier la programmation SIMD sur différentes architectures matérielles.

Alors que la première version (MIPPv1) était écrite directement en C++ en s'appuyant sur la spécialisation de *templates*, la seconde version (MIPPv2), actuellement en développement, repose sur un générateur automatique écrit en Python. Ce générateur exploite le moteur de rendu Jinja2 [23] ainsi qu'un mini-langage de *templating* dédié pour produire l'ensemble des fichiers d'en-tête C/C++, les suites de tests unitaires et la documentation technique.

MIPPv2 s'articule autour d'une architecture en couches (Fig. 9). Chaque couche est fonctionnellement complète (*feature-complete*), garantissant l'accès à l'ensemble des fonctionnalités (types vectoriels, registres, masques, prédication) dès la couche la plus basse. Les couches supérieures réutilisent ces définitions tout en exploitant les mécanismes du langage pour enrichir le niveau d'abstraction et la productivité du développeur :

- la couche C99, propose une interface ressemblant à de la programmation avec des intrinsèques agnostique du jeu d'instructions.
- la couche C++, utilisant la surcharge de fonctions et la spécialisation de *templates* pour unifier les noms des fonctions et des types.
- la couche C++ objet, encapsulant les registres et masques dans des classes ($\text{Rvd}<\text{T}>$, $\text{Rvm}<\text{T}>$) et fournissant du sucre syntaxique via la surcharge des opérateurs arithmétiques (+, -, *, /).

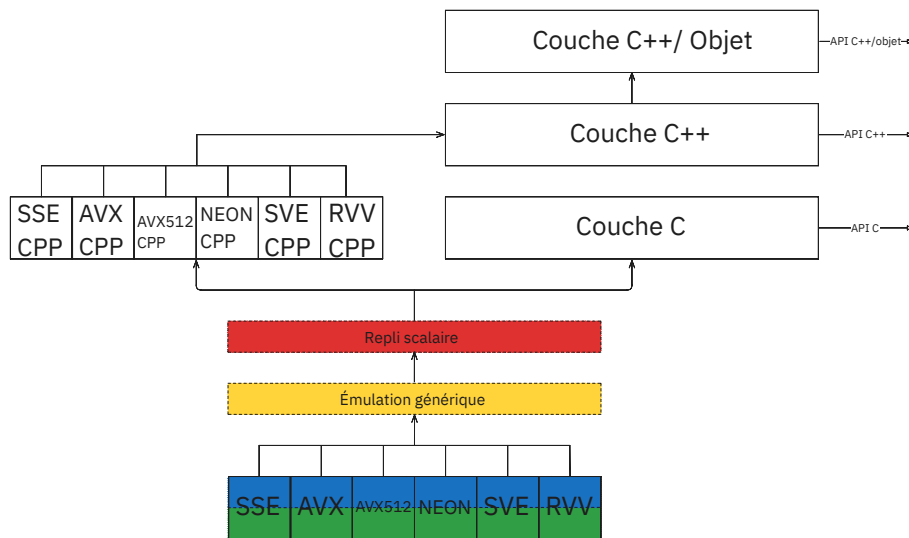


Fig. 9 – Architecture en couches de la bibliothèque MIPP.

Pour implémenter une opération donnée sur une cible matérielle, MIPPv2 distingue quatre niveaux d'implémentation :

- l'implémentation **native** : un appel direct à une fonction intrinsèque correspondant à une instruction assembleur dédiée de l'ISA.
- l'implémentation **émulée-native** : une séquence composée de plusieurs intrinsèques spécifiques à cette même ISA pour pallier l'absence d'instruction unitaire.
- l'implémentation **émulée-générique** : une composition d'appels à d'autres fonctions MIPP, permettant une réutilisation portable d'algorithmes vectoriels entre architectures.
- le **repli scalaire** (*scalar fallback*) : une implémentation scalaire séquentielle garantissant l'exécution sur toute cible matérielle et servant d'oracle de référence lors des phases de test.

Afin de générer une fonction MIPP, on part d'un prototype. Le prototype d'une fonction contient son type de retour, le type et le nombre de ses arguments, ainsi que l'accessibilité de ces derniers en lecture ou en écriture. La définition d'une fonction permet de connaître, entre autres, son prototype et les types pour lesquels la fonction doit être définie (Liste 1).

```
1 protos = {
2   "ret_reg_2args_reg": {
3     "ret": {"type": "reg", "charac": "W0"},
4     "args": [{"type": "reg", "charac": "R0"},
5               {"type": "reg", "charac": "R0"}]
6   },
7 }
```

Python

```
1 mipp_funcs = {
2   "add": { "proto": protos["ret_reg_2args_reg"],
3           "datatypes": all_datatypes },
4   "div": { "proto": protos["ret_reg_2args_reg"],
5           "datatypes": [float32, float64]},
6 }
```

Python

Liste 1 – Exemple d'une définition de prototype (à gauche) et de fonctions MIPP (à droite).

À l'aide de ces définitions abstraites, chaque ISA doit définir une ou plusieurs implémentations de chaque fonction MIPP. À noter qu'une ISA peut définir plusieurs implémentations d'une même fonction MIPP si seulement une extension de cette ISA supporte la fonction. Par exemple, l'extension FMA de AVX2 définit des intrinsèques effectuant un `fmadd`. Ces intrinsèques ne sont pas supportés par AVX2 sans l'extension FMA.

```
1 # une définition d'fmadd en AVX
2 "fmadd": [
3 {
4   "instr_name": "fmadd", # Nom de l'instruction correspondante en AVX
5   "datatypes": all_float, # [float64, float32]
6   "template": tpl_implem_avx["arith_3args"], # Template ci dessous
7   "if": "defined(__FMA__)" # Défini si la macro __FMA__ existe
8 } ],
```

Python

```
1 # Template pour les fonctions arithmétiques à 3 arguments en AVX
2 "arith_3args":
3 {
4   "format": "short",
5   "code": "{{isa.prefix}}_{{instr_name}}_{{isa_dt_par.data_ext}}(r0.r, r1.r, r2.r);",
6 }
```

Python

```

1 // code généré pour cette implémentation d'fmadd
2 #include <immintrin.h> // header qui contient les built-ins AVX
3
4 #if defined(__FMA__) // la fonction est définie si __FMA__ est défini
5 static inline // force le compilateur à remplacer les appels de fonc. par le code
6 rvd_avx_float64_t mipp_avx_fmadd_float64(const rvd_avx_float64_t r0,
7                                           const rvd_avx_float64_t r1,
8                                           const rvd_avx_float64_t r2) {
9     rvd_avx_float64_t res;
10    res.r = _mm256_fmadd_pd(r0.r, r1.r, r2.r); // appel à la fonction intrinsèque AVX
11    return res;
12 }
13 #endif

```

C++ Header

Liste 2 – Exemple d’une définition de la fonction `fmadd` de MIPP en AVX. De haut en bas, les extraits de code correspondent à un *template* de la fonction `fmadd`, au prototype qu’elle utilise et au code C généré.

La Liste 2 montre la génération de la fonction `fmadd` en AVX lorsque le processeur supporte l’extension FMA. Si ce n’est pas le cas, il reste néanmoins nécessaire d’implémenter `fmadd` dans MIPP afin d’assurer la portabilité du code. On propose donc une seconde implémentation de `fmadd`, qui sera émulée (Liste 3).

```

1 "fmadd": {
2     "format": "long",
3     "code": """
4         %r<tp>% rmul = %mul<tp>%(r0, r1);
5         %r<tp>% res = %add<tp>%(rmul, r2);
6         return res;
7     """ },

```

Python

```

1 #if !( defined(__FMA__) ) // défini uniquement si __FMA__ n'est pas défini
2 static inline
3 rvd_avx_float64_t mipp_avx_fmadd_float64(const rvd_avx_float64_t r0,
4                                           const rvd_avx_float64_t r1,
5                                           const rvd_avx_float64_t r2) {
6     rvd_avx_float64_t rmul = mipp_avx_mul_float64(r0, r1); // rmul = r0 * r1
7     rvd_avx_float64_t res = mipp_avx_add_float64(rmul, r2); // res = rmul + r2
8     return res; // res = r0 * r1 + r2
9 }
10 #endif

```

C Header

Liste 3 – De haut en bas, *template* émulé d’`fmadd` et son implémentation générée en AVX.

À l’instar des fonctions, les types de données de MIPP pour les registres de données (`rvd`) et les registres de masques (`rvm`) encapsulent les types natifs de chaque ISA. Lorsqu’une architecture ne fournit pas de registres de masques dédiés (comme AVX2

ou NEON), ceux-ci sont émulsés à l'aide de registres vectoriels conventionnels. L'en-tête `C mipp.h` réalise ensuite l'aiguillage statique en associant les symboles génériques aux types et fonctions spécifiques de l'ISA ciblée (un extrait de ce renommage est donné en annexe Liste 15). La couche C++ de MIPP repose sur un principe similaire, mais exploite la surcharge de fonctions et la spécialisation de *templates* afin de proposer des noms d'opérations courts et uniformes (comme illustré en annexe Liste 16). Enfin, la couche C++ objet encapsule les registres dans des classes (`Rvd<T>`, `Rvm<T>`) et propose des surcharges d'opérateurs arithmétiques pour simplifier l'écriture des algorithmes vectoriels.

2.6.1 | Positionnement de MIPPv2

MIPPv2 s'inscrit dans la continuité de MIPPv1 en conservant sa philosophie fondamentale : fournir une interface unifiée permettant d'écrire un code vectorisé portable sans surcoût à l'exécution. Cette nouvelle version introduit plusieurs évolutions majeures :

- **L'élargissement du support architectural et l'intégration des spécificités matérielles** : MIPPv2 intègre le support complet de RVV et SVE. La bibliothèque applique le principe : « exploiter nativement quand cela est possible, émuler sinon ». Ainsi, les concepts de groupement de registres (LMUL), de division de registres (LDIV) et de prédication vectorielle sont supportés nativement sur les architectures compatibles (RVV, AVX-512, SVE) et émulsés de manière transparente sur les autres cibles. Le Tableau 5 résume le niveau de support natif selon l'ISA.
- **La coexistence d'extensions multiples au sein d'un même programme** : Contrairement à MIPPv1 où une seule extension était active globalement, MIPPv2 permet de manipuler simultanément des registres scalaires, SSE, AVX2 et AVX-512 au sein du même code source via des types explicitement qualifiés.
- **L'introduction d'une interface C99 complète** : En plus des interfaces C++, MIPPv2 expose l'intégralité de ses fonctionnalités via une API C99.
- **Un modèle de taille de vecteur fixée à la compilation** : Afin de garantir l'uniformité sémantique et la compatibilité d'interface avec les architectures à registres fixes (x86, NEON), MIPPv2 n'adopte pas un modèle purement agnostique de la longueur (VLA). La largeur des vecteurs logiques est fixée statiquement à la compilation en fonction de la taille physique des registres de la cible (VLEN).

3 | Travaux effectués

Les contributions apportées à la bibliothèque MIPP s'articulent autour de trois axes principaux : les refontes structurelles du générateur d'en-têtes, l'intégration de l'extension RVV 1.0 et l'enrichissement fonctionnel de la bibliothèque.

3.1 | Changements structurels au générateur d'en-têtes MIPP

L'intégration de RVV 1.0 a nécessité une refonte du générateur d'en-têtes afin de prendre en charge trois mécanismes fondamentaux :

1. le multiplicateur de registres (**LMUL**), permettant d'agréger plusieurs registres physiques au sein d'un vecteur logique,
2. la division de registres (**LDIV**), permettant de manipuler des fractions de registres physiques,
3. les opérations vectorielles prédiquées, appliquant sélectivement les calculs selon un masque booléen.

Fidèle aux principes de conception de **MIPP**, le générateur exploite ces mécanismes de façon native dès que le matériel le permet (**RVV**, **AVX-512**, **SVE**) et génère automatiquement une couche d'émulation pour les autres cibles (**AVX2**, **SSE**, **NEON**).

Concernant le support du **LMUL**, il a été nécessaire de compléter la couche d'émulation pour les opérations retournant une valeur scalaire (telles que l'extraction d'élément **get** ou la réduction horizontale **hadd**). Sur les architectures ne supportant pas nativement le groupement de registres, un vecteur logique à $LMUL > 1$ est représenté par une structure agrégeant plusieurs registres physiques. Un jeu de fonctions génériques a donc été développé pour itérer sur ces sous-registres et agréger leurs résultats scalaires de manière transparente.

Les opérations vectorielles prédiquées ont également été intégrées au générateur **MIPP**. Afin d'offrir une abstraction portable tout en préservant l'expressivité native des architectures matérielles, les spécifications de masquage de **SVE**, **RVV** et **AVX-512** ont été analysées et synthétisées en trois motifs (*patterns*) de masquage (Liste 4) :

- **masks** (fusion avec registre source explicite) : pour chaque élément, si le masque est actif (*masked-on*), la valeur calculée par l'opération est assignée ; sinon, la valeur provient de l'élément correspondant d'un registre source (**rsrc**) passé en paramètre. Ce motif est natif en **RVV** et **AVX-512**.
- **mask** (conservation du premier opérande) : si le masque est inactif, l'élément conserve la valeur du premier registre opérande (**r0**). Ce motif correspond au comportement par défaut d'**ARM SVE**.
- **maskz** (mise à zéro / *zeroing*) : si le masque est inactif, l'élément est forcé à zéro. Ce motif est supporté nativement en **AVX-512**.

```

1  rvd<T> mask_op(rvm<T> m, rvd<T> r0, rvd<T> r1){
2      rvd<T> ret;
3      for(size_t i = 0 ; i < N<T>(); i++)
4          ret[i] = m[i] ? op(r0[i], r1[i]) : r0[i];
5      return ret;
6  }
7
8  rvd<T> maskz_op(rvm<T> m, rvd<T> r0, rvd<T> r1){
9      rvd<T> ret;
10     for(size_t i = 0 ; i < N<T>(); i++)
11         ret[i] = m[i] ? op(r0[i], r1[i]) : 0;
12     return ret;
13 }
14
15 rvd<T> masks_op(rvm<T> m, rvd<T>, rsrc, rvd<T> r0, rvd<T> r1){
16     rvd<T> ret;
17     for(size_t i = 0 ; i < N<T>(); i++)
18         ret[i] = m[i] ? op(r0[i], r1[i]) : rsrc[i];
19     return ret;
20 }

```

Liste 4 – Sémantique séquentielle des trois variantes masquées d’une opération `op` binaire.

Pour intégrer ces trois motifs, les prototypes de fonctions de **MIPP** ont été enrichis d’un attribut précisant les variantes de masquage autorisées. Par exemple, les opérateurs arithmétiques comme `add` ou `mul` supportent tous les masques. Les fonctions effectuant un accès mémoire, comme `load` ou `set1`, ne supportent que `masks` et `maskz`. Les réductions comme `hadd` ou `hmax` ne supportent que `mask`.

Sur les architectures disposant du motif `masks` (comme **RVV**), l’émulation des motifs `mask` et `maskz` s’obtient directement par substitution du registre source (Liste 5).

```

1  rvd<T> mask_op(rvm<T> m, rvd<T> r0, rvd<T> r1) {
2      return masks_op(m, r0, r0, r1); // Le registre source est r0
3  }
4  rvd<T> maskz_op(rvm<T> m, rvd<T> r0, rvd<T> r1) {
5      return masks_op(m, set0<T>(), r0, r1); // Le registre source est nul
6  }

```

Liste 5 – Sémantique de l’émulation des motifs de masquage `mask` et `maskz` avec `masks`.

Sur les architectures dépourvues de masquage matériel (**AVX2**, **SSE**, **NEON**), l’ensemble des motifs est émulé de manière générique à l’aide des primitives `blend`, `andb` et `toreg` (Liste 6).

```

1  rvd<T> mask_op(rvm<T> m, rvd<T> r0, rvd<T> r1){
2      rvd<T> r;
3      rvd<T> r_op = op(r0,r1);
4      return blend(m,r_op, r0);
5  }
6
7  rvd<T> maskz_op(rvm<T> m, rvd<T> r0, rvd<T> r1){
8      rvd<T> r_op = op(r0,r1);
9      rvd<T> r_msk = toreg(m); // r_msk[i] = m[i] ? V_SATURE_T : 0;
10     return andb(r_msk,r_op);
11 }
12
13 rvd<T> masks_op(rvm<T> m, rvd<T>, rsrc, rvd<T> r0, rvd<T> r1){
14     rvd<T> ret_op = op(r0,r1);
15     return blend(m,r_op,rsrc);
16 }

```

Liste 6 – Émulation générique des motifs de masquage.

Ces observations ont permis au générateur MIPP d'émettre des implémentations masquées complètes et fonctionnelles pour l'intégralité des ISA supportées.

3.2 | Implémentation de RVV dans MIPP

RVV a été intégré de manière itérative dans MIPP. D'abord le suport des fonctions avec $LMUL = 1$, puis les groupements étendus ($LMUL \in \{2, 4, 8\}$), les divisions de registres (LDIV), et enfin l'ensemble des variantes prédiquées.

L'une des forces majeures de l'extension RVV 1.0 réside dans l'**orthogonalité** et la régularité de son jeu d'instructions : chaque opération arithmétique ou logique est systématiquement définie pour l'intégralité des types de données (int8 à int64, uint8 à uint64, float32, float64) et des facteurs LMUL. À l'inverse, l'architecture x86 souffre d'hétérogénéités historiques (par exemple, l'opération de multiplication sur entiers 8 bits `_mm256_mul_epi8` est absente d'AVX2, alors que `__riscv_vmul_vv_i8m1` existe nativement en RVV). Cette complétude a permis de concevoir des modèles de code (*templates*) Python hautement factorisés dans le générateur MIPP (Liste 7).

```

1 # Modèles génériques d'implémentation RVV
2 tpl_implem_rvv = {
3     "load": {
4         "format": "short",
5         "code":    "{{prefix}}_vle{{width}}_v_{{data_ext}}({{to_ptr}}*) p0, %N<tp>);"
6     },
7     "arith_2args": {
8         "format": "short",
9         "code":    "{{prefix}}_v{{instr_name}}_vv_{{data_ext}}(r0.r, r1.r, %N<tp>);"
10    },
11 }

```

Python

```

1 # Dictionnaire d'instanciation des opérations MIPP en RVV
2 implems_rvv = {
3     "add": [
4         { "instr_name": "fadd", "datatypes": all_float, "template": tpl_implem_rvv["arith_2args"] },
5         { "instr_name": "add", "datatypes": all_int_uint, "template": tpl_implem_rvv["arith_2args"] }
6     ],
7     "load": [
8         { "instr_name": "vle", "datatypes": all_datatypes, "template": tpl_implem_rvv["load"] }
9     ],
10 }

```

Python

Liste 7 – Modèle générique (`tpl_implem_rvv`) et dictionnaire d'association (`implems_rvv`) pour les opérations `add` et `load` en RVV.

Certaines fonctions requièrent néanmoins une composition d'instructions plus complexe. C'est notamment le cas de la réduction multiplicative horizontale `mipp::hmul`, qui calcule le produit de tous les éléments d'un registre. L'ISA RVV 1.0 intégrant des instructions de réduction pour l'addition (`vfredusum`, `vredosum`) ou les extrema (`vfredmax`, `vredmin`), mais aucune instruction unitaire pour le produit, il a été nécessaire de développer un arbre de réduction logarithmique fondé sur des décalages vectoriels successifs.

À l'issue de ces travaux, la couverture de RVV 1.0 au sein de MIPP est exhaustive : toutes les opérations supportables matériellement sont implémentées nativement, avec une prise en charge complète des facteurs LMUL, LDIV et des motifs de masquage.

3.3 | Enrichissement de MIPP

En complément de l'intégration de RVV 1.0, plusieurs contributions majeures ont été apportées à l'architecture globale et à la couverture fonctionnelle de MIPP.

3.3.1 | Évolutions fonctionnelles et architecturales

Des fonctions manquantes pour SSE, AVX, AVX-512 et NEON ont été implémentées, et de nouvelles primitives vectorielles (telles que `mipp::abs` ou `mipp::cos`) ont été intégrées à l'interface standard.

Un mécanisme de coexistence multi-extensions a également été développé. Particulièrement avantageux sur architecture x86, il permet au développeur d'instancier et de manipuler conjointement des registres scalaires, SSE, AVX et AVX-512 au sein d'un même fichier source via des types explicitement qualifiés.

Par ailleurs, les mécanismes d'émulation ont été généralisés : le générateur permet désormais de définir des modèles d'émulation génériques exprimés sous forme de compositions d'autres fonctions MIPP. Ces modèles sont totalement agnostiques de l'extension SIMD matérielle, réduisant considérablement l'effort d'implémentation pour chaque nouvelle ISA.

Une cible scalaire a également été introduite dans MIPP. Il s'agit d'une « pseudo-ISA » dont les registres sont modélisés par des tableaux séquentiels classiques, sans recourir à des extensions vectorielles matérielles. Cette cible sert de vérité terrain (*ground truth*) pour spécifier la sémantique de référence de MIPP. La largeur des registres scalaires est configurable à la compilation (via les options `-DMIPP_SCALAR` et `-DMIPP_SCALAR_SIZE=X`), s'alignant par défaut sur la taille des registres vectoriels de la cible hôte.

Cette cible scalaire constitue le socle du mécanisme de repli scalaire (*scalar fallback*) : lorsqu'une fonction MIPP ne dispose ni d'implémentation native ni d'émulation générique pour l'ISA active, le générateur émet automatiquement une fonction de repli qui convertit les registres matériels en structures scalaires via `memcpy`, exécute l'opération séquentielle, puis réinjecte le résultat (Liste 8).

```

1  #if !defined(__SSE2__)
2  static inline rvm_sse_float64_t mipp_sse_cmpeq_float64(const rvd_sse_float64_t
   r0, const rvd_sse_float64_t r1) {
3      rvd_scalar_float64_t s_r0 = {0};
4      memcpy(&s_r0, &r0, sizeof(r0));
5      rvd_scalar_float64_t s_r1 = {0};
6      memcpy(&s_r1, &r1, sizeof(r1));
7      rvm_scalar_float64_t sres = mipp_scalar_cmpeq_float64(s_r0, s_r1);
8      rvm_sse_float64_t res;
9      memcpy(&res, &sres, sizeof(res));
10     return res;
11 }
12 #endif

```

Liste 8 – Repli scalaire de la fonction `cmpeq` de MIPP pour les nombres flottants double précision en SSE.

3.3.2 | Générateur de tests

Afin de valider rigoureusement chaque implémentation, un générateur automatique de tests a été intégré à MIPP. Sa portée est universelle : il couvre les cibles SSE, AVX2, AVX-512, NEON et RVV ainsi que toutes les combinaisons de types de données, de masquages et de facteurs LMUL.

Comme les fichiers d'en-tête sont générés à partir d'une base de données formelle de prototypes, le générateur dispose nativement de toutes les métadonnées requises pour synthétiser les cas de test. Développé en Python avec Jinja2, il émet des suites de tests unitaires pour C et C++ adossées au framework de test Catch2 [24].

La méthodologie de test repose sur l'ISA scalaire utilisée comme oracle de référence sémantique : pour chaque opération et chaque configuration de type, le résultat renvoyé par l'ISA matérielle cible est comparé élément par élément à celui produit par l'implémentation scalaire (Liste 9). Chaque test est répété N fois avec des vecteurs d'entrée aléatoires afin de maximiser la couverture des valeurs et la probabilité de détecter des bogues.

```
1 rvd<T>          res  = mipp::add(r1, r2); // ISA par défaut
2 // => dépend de l'architecture sur laquelle le test est compilé
3 rvd<T,1,SCALAR> s_res = mipp::add(s1, s2); // ISA scalaire explicite
4
5 for(size_t i = 0 ; i < mipp::N<T>(); i++)
6     REQUIRE(mipp::get(res,i) == mipp::get(s_res,i))
```

C++

Liste 9 – Principe de validation unitaire comparant l'implémentation vectorielle à l'oracle scalaire.

Pour garantir la robustesse des tests sans faux positifs, les domaines de génération aléatoire et les assertions ont été adaptés à plusieurs spécificités matérielles :

- les fonctions arithmétiques flottantes complexes (`exp`, `cos`, approximations de division) intègrent des seuils de tolérance relative (ϵ), prenant en compte les variations d'arrondi admises par les différentes ISA.
- les réductions horizontales sur registres larges ($LMUL \in \{4, 8\}$) ou types étroits (`int8`, `uint8`) peuvent provoquer des dépassements d'entiers (*overflows*) dont le comportement n'est pas strictement spécifié en C/C++ : des modules utilitaires dédiés ont été développés pour borner les tirages aléatoires et éviter les débordements parasites.

L'automatisation des tests a permis la mise en place d'un pipeline d'intégration continue (CI) sur le dépôt GitLab du projet. À chaque *commit* :

1. la bibliothèque MIPP et les suites de tests unitaires sont générées,
2. l'analyse statique du code est exécutée via `cxxcheck`,
3. les tests sont compilés avec GCC et Clang pour l'ensemble des architectures cibles,

4. les exécutables de test sont validés sur les nœuds du cluster **Dalek** de l'équipe ALSOC du LIP6.

3.3.3 | Générateur de documentation

Un générateur automatique de documentation a été développé pour synthétiser les spécifications et le statut d'implémentation de la bibliothèque sous forme de pages **Markdown**. Ce générateur produit deux types de documents :

- des matrices de couverture récapitulant, pour chaque cible **ISA**, le statut de chaque fonction selon le type de données considéré (Tableau 1) :
 - ✓✓ : implémentation **native** (correspondance directe avec un intrinsèque unitaire),
 - ✓ : implémentation **émulée-native** (séquence d'intrinsèques de la même **ISA**),
 - ✓ : implémentation **émulée-générique** (composition d'autres fonctions **MIPP**),
 - : opération non définie dans la spécification **MIPP** pour le type de données.
- des fiches descriptives unitaires par fonction, intégrant les signatures **C** et **C++**, une description textuelle, ainsi que le pseudo-code de référence issu de l'**ISA** scalaire.

Tableau 1 – Reproduction de la matrice de couverture des fonctions arithmétiques pour l'extension **RVV** dans la documentation **MIPP**.

Function	f64	f32	i64	i32	i16	i8	u64	u32	u16	u8
add	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
adds	–	–	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
sub	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
subs	–	–	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
mul	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
div	✓✓	✓✓	–	–	–	–	–	–	–	–
fmadd	✓✓	✓✓	–	✓✓	–	–	–	–	–	–
fmaddi	✓✓	✓✓	–	✓✓	–	–	–	–	–	–
fmsub	✓✓	✓✓	–	✓	–	–	–	–	–	–
fnmadd	✓	✓	–	✓	–	–	–	–	–	–
fnmsub	✓	✓	–	✓	–	–	–	–	–	–
div2	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
div4	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓

Les fichiers **Markdown** générés sont ensuite compilés à l'aide de l'outil **MkDocs** pour produire un site web statique interactif, constituant la documentation de référence de **MIPPv2**.

4 | Mesures de performances & intégration à des applications HPC

Terminologie et méthode d'évaluation

Dans cette section, le terme *kernel* désigne un noyau de calcul et *runtime* le temps d'exécution d'un programme. Le *speedup* représente le rapport entre le temps d'exécution d'une version de référence et celui d'une version optimisée :

$$\text{speedup} = \frac{T_{\text{référence}}}{T_{\text{optimisée}}}$$

Un *speedup* supérieur à 1 indique une amélioration des performances. Le débit, exprimé en GFLOP/s (milliards d'opérations flottantes par seconde), permet de comparer l'efficacité de calcul de différentes implémentations en normalisant les performances par rapport au volume d'opérations exécutées.

Les benchmarks ont été compilés avec gcc 13.3.0 sur Zen4, avec gcc 14.2.1 et clang 21.1.7 sur Skylake, ainsi qu'avec gcc 15.2.0 et clang 21.1.8 sur X100. Lorsque le compilateur n'est pas explicitement indiqué, gcc est utilisé par défaut. Pour le benchmark de **micro-kernels** GEMM, les légendes des figures précisent le compilateur employé pour chaque variante (gcc ou clang).

Afin de comparer les différentes stratégies de vectorisation dans des conditions rigoureusement homogènes, plusieurs classes d'options de compilation ont été définies pour gcc et clang :

- Version scalaire de référence : -O3 -fno-tree-vectorize -fno-unroll-loops
- Options communes : -O3 -ffast-math -funroll-loops -finline-functions
- Options SSE : -msse4.2 -msse4.1 -msse3 -mssse3 -msse2 -msse
- Options AVX2 : -mavx2 -mfma
- Options AVX-512 : -mavx512f -mavx512vl -mavx512dq -mavx512bw
- Option RVV : -march=rv64gcv_zvl256b -mrvv-vector-bits=zvl
- Ciblage natif : -march=native

Dans les benchmarks unitaire et le benchmark pseudo solveur, métriques matérielles ont été collectées avec *likwid-perfctr* :

- 5.5.2 sur Zen4
- 5.5.1 sur Skylake

Dans le benchmark de produit matriciel, les métriques sont relevées sans outil externe : le temps d'exécution est mesuré à l'aide des utilitaires de l'en-tête standard *<chrono>* et le débit en GFLOP/s est calculé analytiquement.

Les mesures ont été réalisées sur trois des plateformes présentées en Section 2 : Zen4, Skylake et X100. La confrontation des cibles x86 et RISC-V permet d'évaluer MIPP sur des microarchitectures de générations distinctes et reposant sur des ISA fondamentalement différentes. Cette diversité permet de distinguer les comportements intrinsèques à une microarchitecture des tendances généralisables. Par ailleurs, Skylake offre un support complet de l'outil *likwid* [25], tandis que X100 n'en dispose pas. En conséquence, certaines métriques matérielles détaillées (comme le taux de vectorisation) n'ont pas pu être collectées sur X100. Pour caractériser finement cette dernière plateforme, le benchmark de produit matriciel présenté en Section 4.0.3 explore ainsi différentes approches d'optimisation bas niveau.

L'exigence fondamentale à laquelle doit répondre MIPP est de fournir une abstraction à coût nul (*zero-cost abstraction*) : l'écriture d'un *kernel* avec MIPP ne doit induire aucun surcoût mesurable par rapport à une implémentation reposant directement sur des fonctions intrinsèques. Cette propriété doit être vérifiée pour chaque ISA ciblée. Dans le domaine du HPC, où l'efficacité d'exécution prévaut sur les gains d'expressivité, l'adoption de la bibliothèque ne peut se justifier que si cette promesse d'efficacité est rigoureusement tenue.

Afin d'évaluer empiriquement si cette exigence est respectée, nous avons collaboré avec l'IFPEN (A. Anciaux, R. Gayno et K. Mohamed El Maarouf), qui nous a fourni une suite de benchmarks ainsi que des cas d'usage issus du solveur linéaire MCGSolve.

MCGSolver est une bibliothèque industrielle de résolution de systèmes linéaires développée et maintenue par l'IFPEN. MIPPV1 y est supporté au travers d'une interface dédiée (MIPP.h). Le portage de cette interface vers MIPPV2 (MIPPV2.h) a constitué un banc d'essai concret pour valider la maturité de la bibliothèque et compléter les fonctionnalités manquantes.

Les compilateurs modernes sont l'aboutissement de plusieurs décennies d'ingénierie et sont généralement capables de générer du code vectorisé (c'est-à-dire utilisant des instructions SIMD) d'excellente qualité sur de nombreux *kernels*. Il est donc nécessaire d'étudier dans quelles situations une implémentation SIMD explicite d'un *kernel* permet d'obtenir de meilleures performances qu'une vectorisation automatique par le compilateur. Cette question ne dépend pas uniquement du calcul réalisé par le *kernel*, mais également du contexte dans lequel celui-ci est compilé et des informations dont dispose le compilateur. L'expression « implémentation SIMD explicite d'un *kernel* » ne préjuge pas sur le choix de l'implémentation SIMD, qui peut être MIPP, des intrinsèques, ou une autre bibliothèque. Cette problématique n'est pas spécifique à MIPP, mais concerne toute approche visant à exploiter le SIMD explicitement. L'un des objectifs majeurs de cette campagne de mesures est donc de formaliser les scénarios où la vectorisation manuelle s'avère pertinente.

Pour mener cette évaluation, trois bancs d'essais complémentaires ont été conçus, chacun explorant un régime calculatoire et une question d'optimisation distincts :

1. **les benchmarks d'algèbre linéaire (régime borné par la mémoire)** : portant sur des opérations **BLAS-1** élémentaires et un pseudo-solveur itératif, ils manipulent des calculs à faible intensité arithmétique. L'objectif est de vérifier l'absence de régression face à MIPPV1 et de quantifier l'absence de surcoût d'abstraction par rapport aux intrinsèques constructeurs et à l'auto-vectorisation,
2. **les benchmarks de lois physiques (régime à divergence de contrôle)** : issus de simulations de réservoir (thermodynamique, perméabilité), ils comportent des branchements conditionnels et des fonctions transcendantes. L'objectif est de déterminer si une restructuration algorithmique légère (masquage **branch-free**, chemins rapides **fastpaths**) permet d'obtenir un gain substantiel là où l'auto-vectorisation du compilateur échoue,
3. **le micro-benchmark de produit matriciel dense GEMM (régime borné par le calcul)** : noyau **BLAS-3** à haute intensité arithmétique, il vise à tester la capacité de MIPPV2 à concevoir des **micro-kernels** capables de saturer le débit crête matériel (> 90 % du pic théorique) et d'explorer efficacement l'espace d'optimisation (blocs de registres, **LMUL**) sur des architectures aux paradigmes vectoriels distincts (**AVX2** et **RVV**).

4.0.1 | Benchmarks d'algèbre linéaire

Dans ce premier banc d'essai, la dimension des vecteurs d'entrée est fixée à dix mille éléments en virgule flottante double précision (**double** en C/C++). Chaque fonction est

exécutée dix mille fois consécutivement. Le temps d'exécution mesuré correspond à la durée totale de cette boucle d'appels.

Les *kernels* considérés implémentent des opérations *BLAS-1* (*Basic Linear Algebra Subprograms*), limitées à des opérations élémentaires vecteur-vecteur ou vecteur-scalaire (addition, multiplication, combinaison linéaire). Les versions MIPP transposent directement les boucles scalaires en instructions vectorielles, sans réorganisation algorithmique avancée. Le code des versions scalaires de ces *kernels* est donné en annexe (Liste 17).

La Fig. 10 met en regard le temps d'exécution et le *speedup* obtenus par la version MIPPv2 par rapport à la version auto-vectorisée pour les différents *kernels* sur architecture Zen4. Ces codes ont été compilés avec les options communes de `gcc`, les extensions `AVX-512` et le ciblage `-march=native`. On constate que, sur les *kernels* *BLAS-1* les plus simples (Scale, Dot et AXPY), la version auto-vectorisée par le compilateur s'avère légèrement plus rapide que la version vectorisée explicitement. En revanche, pour les *kernels* comportant des réductions horizontales (Squared Norm, Dot + Squared Norm et AXPY + Reduction), la version MIPPv2 obtient un gain de performance significatif, avec un *speedup* supérieur à 5.

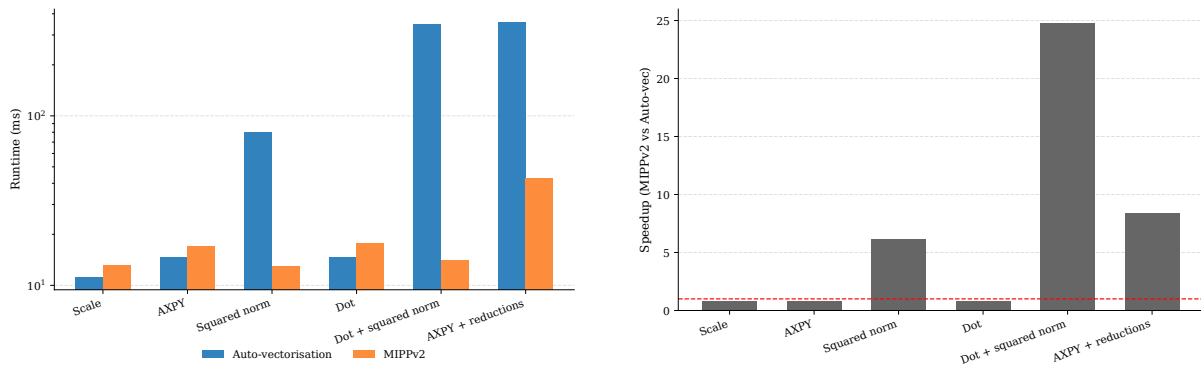


Fig. 10 – Comparaison du temps d'exécution (à gauche) et du speedup (à droite) des *kernels* d'algèbre linéaire unitaires sur Zen4

Afin d'identifier les causes de ces écarts de performance, nous avons évalué le taux de vectorisation des différents *kernels*. Cette métrique matérielle est rapportée par `likwid` sur `Skylake`, mais n'est pas disponible sur `Zen4`.

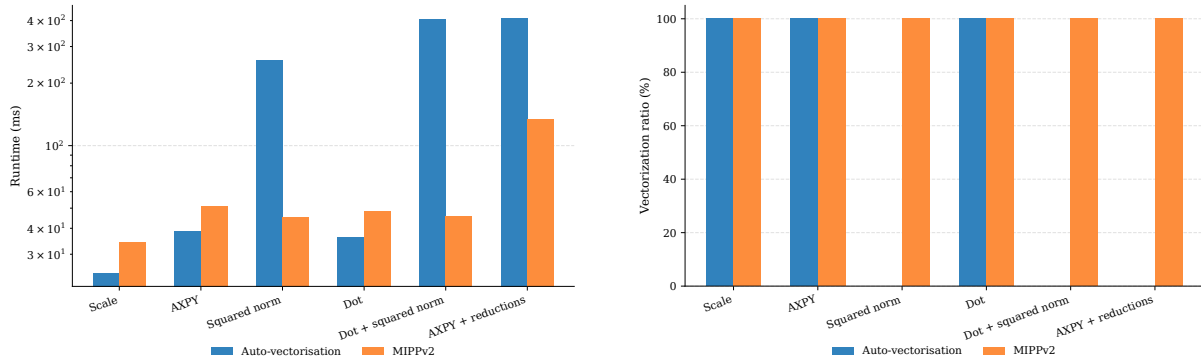


Fig. 11 – Comparaison du temps d'exécution (à gauche) et du taux de vectorisation (à droite) des différents *kernels* sur Skylake.

Sur la Fig. 11, nous rapportons le temps d'exécution des différents *kernels* sur Skylake. Les *kernels* ont été compilés avec gcc avec les options communes, AVX2 et le ciblage `-march=native`.

On retrouve les mêmes tendances que sur Zen4 : un temps d'exécution légèrement supérieur pour la version MIPPV2 sur les *kernels* les plus simples, mais significativement inférieur sur les *kernels* comportant des réductions.

Les mesures rapportées par likwid confirment que le compilateur auto-vectorise intégralement les *kernels* BLAS-1 élémentaires. En revanche, les *kernels* de réduction, pour lesquels la version MIPPV2 apporte un gain massif, ne sont pas vectorisés par le compilateur. L'analyse du taux de vectorisation permet ainsi d'établir un premier cas de figure dans lequel MIPPV2 obtient de meilleures performances : lorsque le compilateur échoue à produire du code vectorisé. Ce banc d'essai unitaire d'algèbre linéaire permet de dégager deux conclusions préliminaires :

1. Il n'est pas toujours pertinent d'écrire explicitement un *kernel* SIMD ; l'auto-vectorisation constitue parfois une solution satisfaisante,
2. Lorsque le compilateur ne parvient pas à auto-vectoriser un *kernel*, une implémentation SIMD explicite est susceptible d'apporter un gain de performance.

Le banc d'essai du pseudo-solveur repose sur les mêmes *kernels* que les benchmarks unitaires. Dans cette configuration, les *kernels* sont exécutés successivement au sein d'une boucle itérative, reproduisant la séquence de calcul de la phase de prétraitement d'un solveur linéaire. Afin d'obtenir des résultats comparables aux mesures unitaires, la dimension des vecteurs et le nombre d'itérations du solveur sont fixés à dix mille. Le code de la boucle principale du pseudo-solveur est présenté en annexe Liste 20.

Ce *benchmark* ainsi que l'interface SIMD générique des différents *kernels* sous-jacents ont été fournis par l'IFPEN. Cette interface est compatible avec deux implémentations : MIPPV2.h, développée dans le cadre de ce stage à partir de l'interface MIPP.h historique, et AVX2.h, une implémentation de référence écrite directement à l'aide d'intrinsèques AVX2 par l'IFPEN.

Dans un premier temps, nous avons comparé les performances du *backend* MIPPV2 à celles du *backend* AVX2 natif. À cette fin, les deux implémentations ont été compilées avec les options communes et les options spécifiques à AVX2. Les mesures confirment que, pour ce programme, MIPPV2 n'introduit aucun surcoût mesurable par rapport à une écriture directe en intrinsèques AVX2.

Nous avons ensuite cherché à déterminer si les autres cibles x86 de MIPPV2 (SSE et AVX-512) présentaient une efficacité équivalente. Plutôt que de développer manuellement des *backends* dédiés utilisant des intrinsèques pour chaque extension, nous avons confronté les temps d'exécution de MIPPV2 à ceux obtenus par auto-vectorisation ainsi qu'à la version MIPPV1 d'origine.

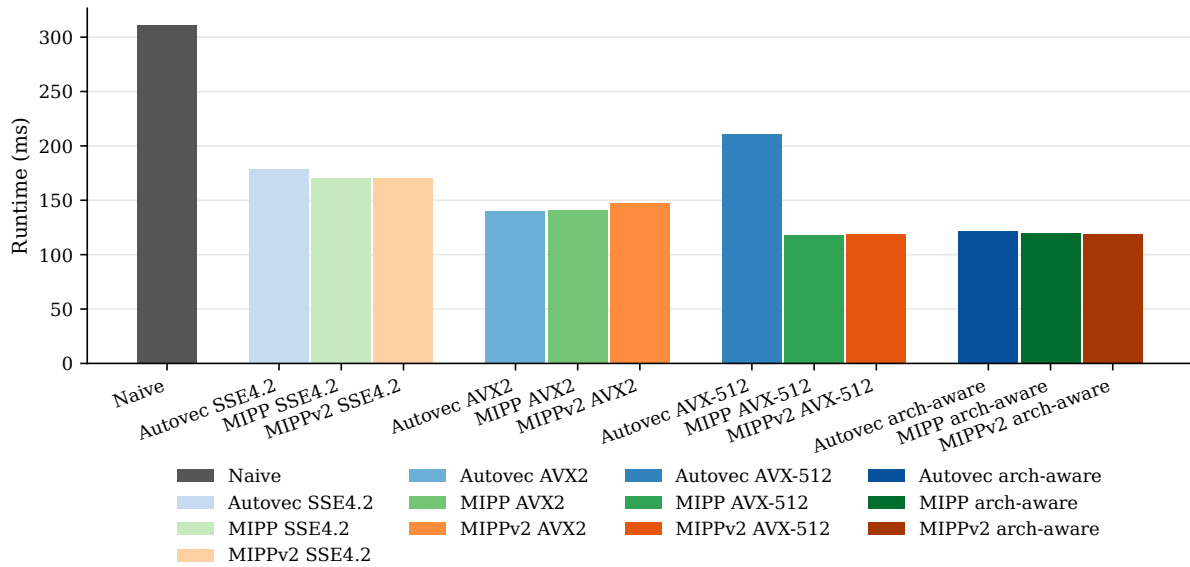


Fig. 12 – Comparaison des temps d'exécution du pseudo-solveur pour les versions MIPP, MIPPV2 et auto-vectorisées selon les extensions SIMD x86 sur Zen4.

La Fig. 12 présente les résultats obtenus sur Zen4. La version naïve, compilée sans vectorisation, affiche le temps d'exécution le plus élevé. Les mesures montrent qu'il n'existe aucun écart significatif de performance entre les versions auto-vectorisées, MIPP et MIPPV2 pour les extensions SSE et AVX2. De même, l'utilisation de l'option `-march=native` (notée *arch-aware* sur la figure) conduit à des performances identiques. Le seul cas où l'auto-vectorisation présente une dégradation notable de performance correspond à la configuration AVX-512 de gcc.

Ces résultats mettent en évidence une divergence notable entre les mesures unitaires et celles du pseudo-solveur. Dans le banc d'essai unitaire, le *kernel* de réduction `Dot + Squared norm` n'avait pas pu être vectorisé par le compilateur, que ce soit en configuration AVX2 ou AVX-512. Or, dans le pseudo-solveur, les temps d'exécution des versions auto-vectorisées varient selon l'extension SIMD ciblée et demeurent très proches de ceux obtenus avec MIPP et MIPPV2. Cela confirme que les *kernels* constitutifs de la boucle ont bien été vectorisés automatiquement par gcc.

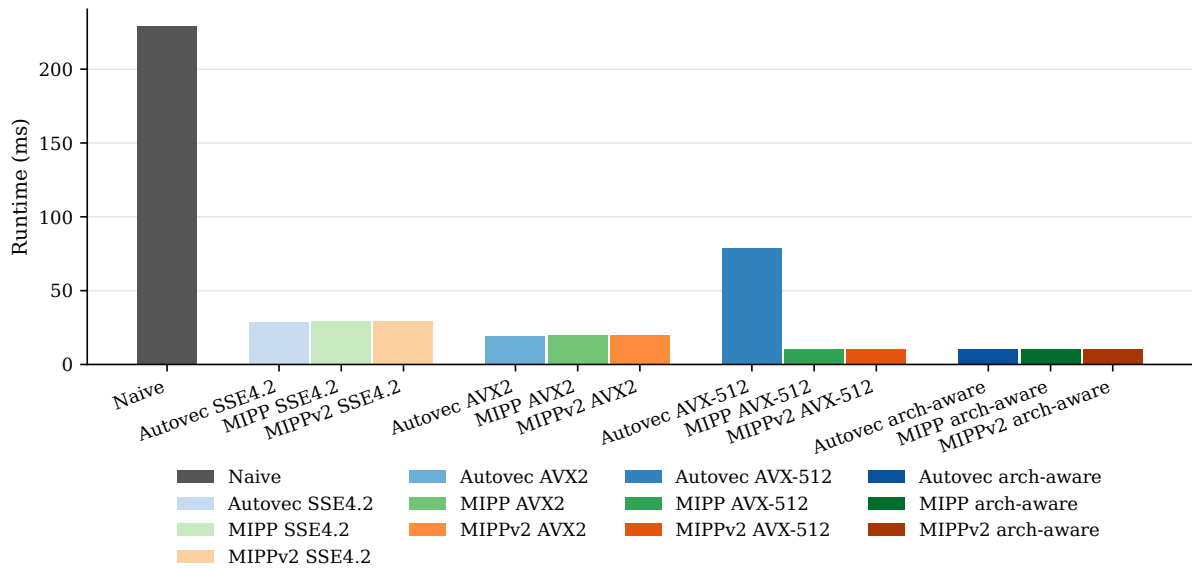


Fig. 13 – Comparaison des temps d'exécution du *kernel Dot + Squared norm* dans le pseudo-solveur pour les versions MIPP, MIPPV2 et auto-vectorisées sur les différentes extensions SIMD x86 sur Zen4.

La figure Fig. 13 tend à confirmer notre hypothèse : le *kernel Dot + Squared Norm* est effectivement vectorisé dans le pseudo-solveur. Si ce n'était pas le cas, le temps d'exécution des *kernels* auto-vectorisés seraient comparables à celui de la version naïve du *kernel*, ce qui n'est pas ce que nous observons.

Nous n'avons pas identifié avec certitude les raisons pour lesquelles le compilateur parvient à vectoriser *Dot + Squared norm* dans le solveur mais échoue dans le banc unitaire. Une différence déterminante réside dans l'allocateur mémoire du pseudo-solveur, qui fournit deux garanties majeures :

- Les tableaux sont alignés sur 64 octets, ce qui garantit un alignement optimal pour les vecteurs AVX-512 (registres de 512 bits) et satisfait *de facto* les contraintes de SSE et AVX2.
- Chaque vecteur est alloué sur une page mémoire distincte, garantissant l'absence stricte de chevauchement (*aliasing*) entre les zones mémoires pointées, de façon analogue à la sémantique du mot-clé *restrict*.

L'une de ces propriétés, ou leur combinaison, aurait permis au modèle de coût et à l'analyse de dépendances du compilateur de valider la transformation vectorielle.

Nous n'avons pas identifié avec certitude pourquoi la version auto-vectorisée avec les options génériques AVX-512 de gcc sur Zen4 n'est pas aussi performante que les versions MIPPV1 et MIPPV2 (bien qu'elle demeure nettement plus rapide que la référence scalaire), alors que cet écart s'efface avec l'option `-march=native` (où l'extension AVX-512 est pourtant bien activée). Une hypothèse plausible réside dans le modèle de coût générique de gcc : sans ciblage microarchitectural précis, le compilateur peut choisir un ordonnancement de boucle ou un schéma de réduction sous-optimal. L'option -

`march=native` permet quant à elle d'activer les réglages propres à l'architecture Zen4 (znver4), rétablissant ainsi un niveau de performance compétitif.

Ce *benchmark* nous a cependant permis d'enrichir notre conclusion préliminaire et de tirer de nouvelles conclusions :

1. Pour un *kernel* donné, il est préférable de ne pas généraliser la capacité du compilateur à auto-vectoriser ou non celui-ci. Il est nécessaire de vérifier expérimentalement si l'auto-vectorisation obtenue fournit des performances satisfaisantes,
2. Lorsqu'elle opère avec succès, l'auto-vectorisation atteint un niveau de performance équivalent à celui de MIPV2, confirmant l'absence de surcoût d'abstraction de la bibliothèque sur x86,
3. Une unique base de code MIPV2 a permis de cibler sans modification les jeux d'instructions SSE, AVX2 et AVX-512, concrétisant la portabilité architecturale visée.

4.0.2 | Benchmarks de lois physiques

En plus des *kernels* d'algèbre linéaire *BLAS-1*, l'IFPEN nous a fourni une suite de *kernels* implémentant différentes lois physiques. Ces *kernels* modélisent le calcul de masse molaire, la loi de pression capillaire, la loi de perméabilité relative et l'équation d'état cubique de Peng-Robinson.

Intuition physique & Contexte applicatif des lois

Dans un simulateur de réservoir pétrolier ou de stockage de CO₂ (ex. Geoxim à l'IFPEN ou GEOSX [26]), le sous-sol est discrétisé en mailles contenant de la roche et des fluides sous pression.

1. Masse molaire

Dans chaque maille, le fluide est un mélange d'espèces (méthane, propane, eau, etc.). Connaître la masse molaire moyenne du mélange permet de relier les quantités de matière aux masses volumiques et aux flux de transport [27].

2. Pression capillaire

Dans les canaux microscopiques de la roche, l'eau et l'huile ne se mélangent pas. La roche attire préférentiellement l'eau (fluide mouillant), ce qui crée un ménisque courbé et une différence de pression entre les deux fluides [28]. Quand les pores sont presque secs, l'effet de succion capillaire est très fort ; à mesure que l'eau remplit les pores, cet effet s'annule.

3 Perméabilité relative

Quand l'eau et le gaz s'écoulent ensemble dans les mêmes pores, ils se gênent mutuellement [29]. La perméabilité relative mesure la part de perméabilité totale accessible à l'eau :

- En dessous d'un certain seuil (S_{wc}), l'eau est coincée dans les culs-de-sac de la roche et ne circule pas ($k_{rw} = 0$).
- Dès que des canaux continus se forment, le débit augmente rapidement selon une loi de puissance non linéaire.

4. Peng-Robinson

Sous haute pression sous terre, les molécules d'hydrocarbures s'attirent et occupent un volume non nul, violant la loi des gaz parfaits [30]. L'équation d'état détermine si le fluide dans la maille se condense en **liquide** (molécules très rapprochées, petit volume V donc racine Z_{min}) ou reste sous forme de **gaz** (molécules espacées, grand volume V donc racine Z_{max}). Le choix physique est dicté par l'état d'énergie minimale de Gibbs $G(Z)$.

Dans chacune de ces lois, les *kernels* calculent la valeur de la fonction ainsi que sa dérivée pour l'ensemble des éléments d'un vecteur. Le *kernel* de masse molaire est le plus élémentaire des quatre : il effectue un produit matrice-vecteur dense (GEMV, opération *BLAS-2*). Les trois autres lois sont nettement plus complexes. Leur implémentation a nécessité d'intégrer des fonctions mathématiques à MIPP et d'adapter la gestion des branchements conditionnels au paradigme SIMD. Le code des versions scalaires de ces lois est donné en annexe (Liste 18, Liste 19).

Les fonctions mathématiques requises comprennent des fonctions transcendantes (puissance, logarithme, exponentielle) et trigonométriques (cosinus, arc-tangente). Afin de les supporter de manière portable dans MIPP, nous avons porté plusieurs routines vectorielles issues de SLEEF [31] et de `avx_mathfun` [32] au sein de la couche d'émulation générique. Ces fonctions sont ainsi immédiatement disponibles pour l'ensemble des ISA supportées.

Dans les lois de perméabilité relative, de pression capillaire et de Peng-Robinson, les calculs dépendent fortement de conditions sur les données d'entrée : régimes de saturation par seuils ($S \leq S_r$ ou $S \geq 1 - S_{r, \text{ref}}$) ou signe du discriminant cubique ($\Delta > 0$ vs $\Delta \leq 0$). En programmation scalaire, ces conditions sont traitées par des embranchements `if/else`. En programmation vectorielle, les différentes voies d'un même registre SIMD peuvent appartenir simultanément à des régimes distincts. Pour gérer cette divergence de flot sans dégrader les performances, deux stratégies complémentaires ont été mises en œuvre :

1. **Vectorisation sans branchement (*branch-free*) par fusion masquée :**
L'ensemble des voies vectorielles évalue les branches de calcul de façon spéculative. Les résultats finaux sont ensuite sélectionnés élément par élément à l'aide de registres de masque et d'instructions de fusion (`mipp::blend`, `mipp::min`, `mipp::max`). Par exemple, dans la loi de perméabilité relative, les discontinuités de saturation effective $\overline{S_w}$ sont résolues via des extrema vectoriels (`min/max`), tandis que les valeurs des dérivées aux limites sont injectées par `mipp::blend`,
2. **Chemins d'exécution rapides vectoriels (*fast paths*) :** L'approche purement *branch-free* impose d'évaluer des fonctions coûteuses (notamment l'exponentiation `pow`) même lorsque certaines voies se trouvent dans un régime trivial où le résultat est connu analytiquement ($k_{rw} = 0$ ou $k_{rw} = 1$). Afin de court-circuiter ces calculs lourds, un test global est appliqué au masque vectoriel : si **toutes** les voies du registre satisfont simultanément la condition triviale (via `mipp::testz_2` sur le masque combiné), l'exécution emprunte un chemin rapide affectant directement les constantes sans évaluer `pow`. Ce mécanisme apporte un gain de performance substantiel dès lors que les mailles présentent une cohérence spatiale (voies vectorielles regroupées dans un même régime physique), au prix d'un léger surcoût lorsque les données sont hétérogènes.

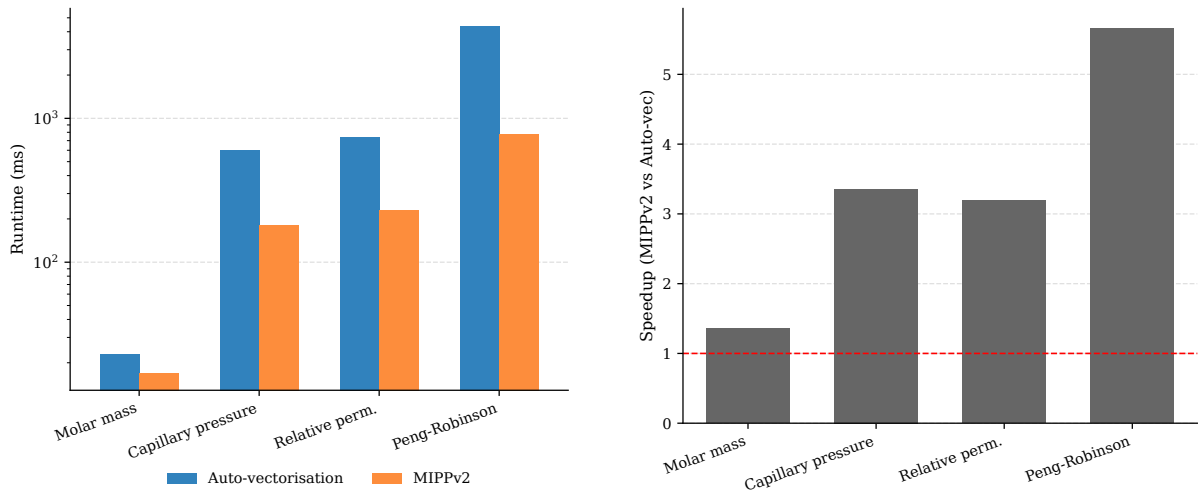


Fig. 14 – Comparaison du temps d'exécution (à gauche) et du speedup (à droite) des différents *kernels* physiques sur Zen4.

Sur la Fig. 14, la version MIPpv2 du *kernel* de masse molaire affiche des performances compétitives avec celles de la version auto-vectorisée. Les lois de pression capillaire et de perméabilité relative atteignent toutes deux un speedup autour de 3. Peng-Robinson obtient le plus grand speedup, atteignant plus de 5. Afin d'identifier l'origine de ces gains, nous avons mesuré le taux de vectorisation avec *likwid* sur Skylake.

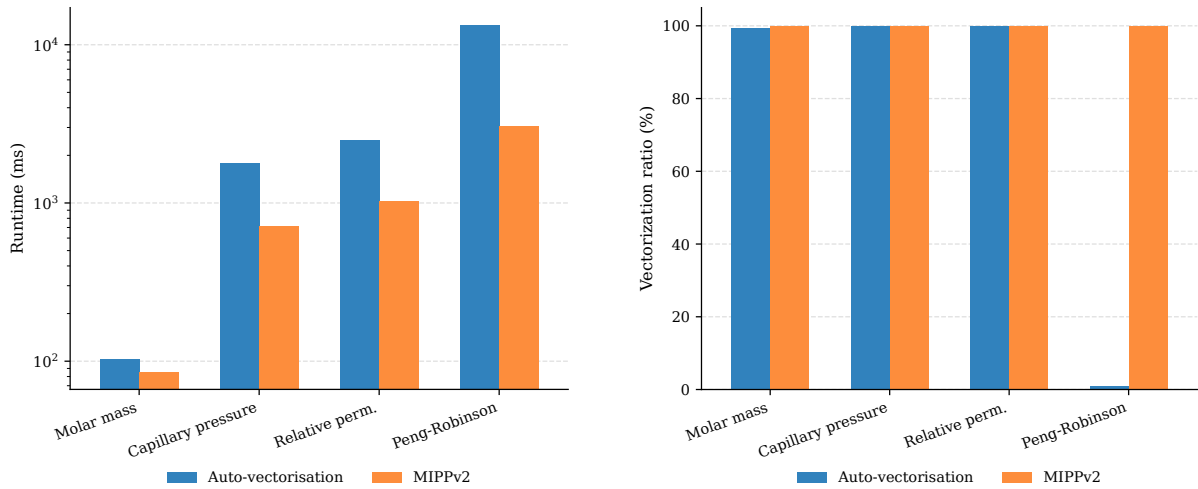


Fig. 15 – Comparaison du temps d'exécution (à gauche) et du taux de vectorisation (à droite) des différents *kernels* physiques sur Skylake.

La Fig. 15 montre que les tendances en temps d'exécution sur Skylake sont identiques à celles de Zen4. Les métriques rapportées par *likwid* révèlent que le compilateur auto-vectorise à près de 100 % la masse molaire, la perméabilité relative et la pression capillaire, mais échoue totalement sur Peng-Robinson (0 % de vectorisation). Nous observons donc que pour les lois physiques de pression capillaire et de perméabilité relative, l'implémentation MIPpv2 présente un gain de performance significatif et ce même

si les versions auto-vectorisées sont pleinement vectorisées par le compilateur. Afin de comprendre ce résultat, rappelons que la métrique fournie par `likwid` quantifie uniquement la proportion d'instructions vectorielles exécutées par le processeur. Lorsque le compilateur vectorise ces lois en s'appuyant sur des routines vectorielles externes (comme `libmvec`), le taux d'instructions SIMD est bien mesuré à 100 %. Néanmoins, cette métrique matérielle ne rend pas compte de l'efficacité intrinsèque des algorithmes d'approximation employés ni des éventuels surcoûts liés aux conventions d'appel de fonctions externes (sauvegardes de registres sur la pile, barrières d'optimisation). L'approche **header-only** inlinee de MIPP permet au compilateur d'optimiser l'ensemble du calcul au sein de la boucle.

Ces *benchmarks* unitaires de physique permettent de confirmer les conclusions précédentes et d'en dégager une nouvelle :

1. Lorsqu'un *kernel* n'est pas auto-vectorisé par le compilateur (cas de Peng-Robinson), une implémentation SIMD explicite permet d'obtenir un gain de performance majeur (ici un *speedup* supérieur à 5),
2. Pour certaines classes de *kernels*, une implémentation SIMD explicite peut obtenir de meilleures performances qu'une version auto-vectorisée, même lorsque le compilateur produit effectivement du code vectorisé. C'est le cas pour la loi de perméabilité relative et la pression capillaire.

Des mesures complémentaires évaluant la sensibilité des performances à la dimension des vecteurs d'entrée et à la précision flottante (simple vs double précision) sont présentées en annexe (Fig. 18, Fig. 19).

4.0.3 | Benchmark micro-kernels GEMM

Rappel — Produit matriciel et HPC

En HPC, *General Matrix Multiply* (GEMM) désigne l'opération fondamentale d'algèbre linéaire dense de la forme : $C = \alpha AB + \beta C$ où A, B, C sont des matrices et α, β des scalaires. En tant que noyau BLAS-3 limité par la puissance de calcul (**compute-bound**), le GEMM constitue l'étalon de référence pour évaluer les capacités de calcul crête d'une microarchitecture. Dans nos benchmarks, nous évaluons la forme canonique $C = AB$ (avec $\alpha = 1, \beta = 0$), où A, B, C sont de dimensions respectives $M \times K, K \times N$ et $M \times N$:

$$\forall i, j : c_{i,j} = \sum_{k=0}^{K-1} a_{i,k} b_{k,j}$$

Une implémentation directe par trois boucles imbriquées (ijk, Liste 10) calcule chaque élément $c_{i,j}$ par le produit scalaire d'une ligne de A et d'une colonne de B .

```
1 static void gemm_ijk(const Matrix<T>& A,
2   const Matrix<T>& B, Matrix<T>& C) {
3   for(size_t i = 0; i < A.rows; ++i)
4     for(size_t j = 0; j < B.cols; ++j) {
5       T sum = 0;
6       for(size_t k = 0; k < A.cols; ++k) sum +=
7         A[i][k] * B[k][j];
8       C[i][j] = sum;
9     }
10 }
```

Liste 10 – Implémentation naïve d'un produit matriciel en C++.

Pour des matrices volumineuses stockées par lignes (**Row-Major**), l'accès à $B[k][j]$ le long de la boucle interne en k s'effectue avec un pas mémoire de N éléments. Cette absence de contiguïté spatiale entraîne des défauts de cache massifs

(**cache misses**). Pour y remédier, les algorithmes performants reposent sur le produit matriciel par blocs :

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Ce partitionnement permet d'adapter la taille des blocs de données aux différents niveaux de la hiérarchie mémoire (caches L3, L2, L1). Les blocs extraits peuvent être réorganisés en mémoire continue avant calcul : cette étape de *packing* garantit des accès vectoriels contigus et alignés. L'architecture d'un GEMM optimisé (Liste 21) est alors articulée en deux niveaux :

- un **macro-kernel**, qui gère le parcours des **blocs de cache** et l'orchestration des données.
- un **micro-kernel**, qui effectue les opérations de multiplication-accumulation (FMA) sur un **bloc de registres** de dimensions $M_R \times N_R$.

Dans le **micro-kernel**, le bloc de registres $M_R \times N_R$ correspond à la sous-matrice de C maintenue directement dans les registres vectoriels (accumulateurs) pendant toute l'itération sur K . Sa largeur N_R est dimensionnée selon la taille des registres SIMD (V_L), et sa hauteur M_R est maximisée pour saturer le banc de registres physiques sans provoquer de déversement mémoire (**spilling**). s disponibles pour maintenir les accumulateurs du micro-kernel.

Dans cette étude, nous nous concentrons sur la conception et l'évaluation de **micro-kernels** de produit matriciel sur les architectures Skylake et X100. L'objectif est de mesurer si MIPPV2 permet d'atteindre le débit arithmétique crête du cœur processeur sans induire de surcoût d'abstraction par rapport à une écriture en assembleur ou en intrinsèques natifs. Afin d'isoler strictement l'efficacité microarchitecturale des boucles de calcul, les mesures sont effectuées sur des blocs de matrices de dimensions calibrées (32×32 à 512×512), préalablement alignés sur 64 octets et packés en mémoire (PackedRowMajor, PackedColMajor).

Dans un premier temps, nous avons implémenté une version vectorielle naïve avec MIPP (Liste 11), transposant directement l'algorithme scalaire ijk (Liste 10) avec un bloc de registres minimal de hauteur $M_R = 1$ et de largeur $N_R = V_L$.

```

1  template<int lmul = 1>
2  static inline void gemm_mippv2(const PackedRowMajor<T>& A,
3                                const PackedRowMajor<T>& B,
4                                PackedRowMajor<T>& C) {
5      constexpr size_t MR = 1;
6      constexpr size_t NR = N<T, lmul>();
7
8      for(size_t i = 0; i < A.rows; i += MR) {
9          for(size_t j = 0; j < B.cols; j += NR) {
10             auto c = set0<T, lmul>();
11             for(size_t k = 0; k < A.cols; ++k) {
12                 const auto a = set1<T, lmul>(A[i][k]);
13                 const auto b = load<T, lmul>(B[k] + j);
14
15                 c = fmadd(a, b, c);
16             }
17             store(C[i] + j, c);
18         }
19     }
20 }
```

Liste 11 – Implémentation d'un **micro-kernel** de GEMM naïf avec MIPP.

Afin d'optimiser ce noyau de calcul, nous avons mené une exploration systématique de l'espace de conception (**Design Space Exploration**), assistée par des outils d'IA générative pour générer et tester rapidement un grand nombre de variantes. Ces variantes explorent le dimensionnement des blocs de registres (M_R, N_R), les valeurs de LMUL, les facteurs de déroulage de la boucle interne k , ainsi que différentes dispositions mémoire. Au terme de cette phase d'exploration, trois classes principales de **micro-kernels** ont été retenues :

1. **Register Blocked (RB)** : Organisation canonique où une sous-matrice de résultats $M_R \times N_R$ est maintenue dans les accumulateurs vectoriels, tandis que les éléments de A sont diffusés (**broadcast**) et que les lignes de B sont chargées vectoriellement le long de K ,
2. **Panel Register Blocked** : Variante exploitant un format de **packing** où A est stocké en colonnes (**Col-Major**) et B en lignes (**Row-Major**), afin d'évaluer le gain apporté par une contiguïté mémoire accrue lors des parcours en K ,
3. **Dot Product** : Approche basée sur le produit scalaire vectoriel où les éléments le long de la dimension K sont multipliés vectoriellement puis réduits horizontalement (`mipp::hadd`).

Contrairement aux approches par blocs de registres, la stratégie **Dot Product** est rarement privilégiée sur CPU en raison de la latence des réductions horizontales, mais elle est courante sur les architectures massivement parallèles (GPU). Son inclusion permet d'évaluer le comportement et la portabilité d'un schéma d'accumulation alternatif sur nos processeurs cibles.

À partir des classes identifiées, nous avons analysé les caractéristiques microarchitecturales du processeur **Skylake** afin de concevoir un **micro-kernel** atteignant le débit crête de la machine. Sur **Skylake**, l'analyse du code généré à partir de la boucle k de la Liste 11 montre que l'assembleur produit prend la forme présentée en Liste 12.

```
1 vbroadcastsd    (%rdi), %ymm0      ; Port 2/3 (Load & Broadcast scalaire de A)
2 vmovapd        (%rsi), %ymm1      ; Port 2/3 (Load vecteur aligné de B)
3 vfmadd231pd    %ymm0, %ymm1, %ymm2 ; Port 0/1 (Multiplication-accumulation dans C)
```

Liste 12 – Code assembleur AVX2 généré pour la boucle k naïve sur Skylake.

Sur l'architecture superscalaire de **Skylake**, les unités de chargement mémoire (ports 2 et 3) et les unités de calcul vectoriel flottant (ports 0 et 1) peuvent fonctionner en parallèle. Cependant, dans cette implémentation naïve, deux facteurs limitent drastiquement les performances :

1. **la chaîne de dépendances de données** : L'instruction `vfmadd231pd` consomme immédiatement les registres `%ymm0` et `%ymm1`, ce qui empêche de masquer la latence des chargements mémoire,

2. **la latence du pipeline FMA** : Sur Skylake, une instruction `vmadd` présente une latence de 4 cycles. Avec un unique accumulateur (`%ymm2`), chaque itération dépend du résultat de la précédente, ce qui limite le débit atteignable.

Pour saturer les unités de calcul, un **micro-kernel** optimal doit exposer suffisamment d'accumulateurs indépendants pour masquer cette latence, tout en exploitant au mieux les 16 registres vectoriels `ymm` disponibles sans provoquer de déversement en mémoire (**register spilling**). Dans nos expériences, la configuration optimale sur Skylake est un **micro-kernel** de la classe **Register Blocked** reposant sur un bloc de registres 4×8 ($M_R = 4$ lignes, $N_R = 8$ colonnes, soit $2 \times V_L$ en double précision). Dans la boucle interne (Liste 22), ce noyau mobilise simultanément 14 registres vectoriels :

- **8 accumulateurs vectoriels** (4×2) maintenant la sous-matrice de résultats.
- **2 registres de chargement** pour les 8 éléments de la ligne courante de B .
- **4 registres de diffusion** pour les 4 éléments de la colonne courante de A .

Ce dimensionnement 4×8 présente un double avantage microarchitectural :

1. **réutilisation des données** : Chaque vecteur chargé de B est réutilisé 4 fois, et chaque valeur diffusée de A est réutilisée 2 fois. Ce ratio réduit la pression sur les ports de chargement mémoire,
2. **parallélisme d'instructions (ILP)** : Les 8 accumulateurs indépendants permettent d'entrelacer 8 instructions `vmadd` consécutives, masquant intégralement la latence de 4 cycles des unités FMA.

Sur X100, les premiers essais ont révélé des performances très faibles, aucun **micro-kernel** n'atteignant 50 % du débit maximal théorique de la plateforme. Pour identifier l'origine de cette limitation, nous avons analysé le code assembleur généré par le compilateur pour la boucle k naïve (Liste 11), présenté en Liste 13 :

```
1 fld      ft0, 0(a_ptr) ; chargement scalaire depuis A (registre f)
2 vfmv.v.f v0, ft0      ; diffusion scalaire -> vecteur (broadcast)
3 vle64.v  v8, (b_ptr)   ; chargement vectoriel depuis B
4 vmadd.vv v16, v0, v8    ; multiplication-accumulation vecteur-vecteur
```

RISC-V

Liste 13 – Code assembleur RVV généré pour la boucle k naïve sur X100.

Dans cette séquence, l'opération de diffusion `mipp::set1` est traduite par un chargement scalaire `fld` suivi d'une instruction vectorielle `vfmv.v.f`. À l'aide de micro-benchmarks synthétiques mesurant le débit de suites d'instructions élémentaires, nous avons observé que la présence d'instructions `vfmv.v.f` au sein d'une chaîne de `vmadd.vv` indépendantes dégrade fortement les performances du **micro-kernel**. Nous faisons l'hypothèse que cette limitation provient du fait que `vfmv.v.f` et `vmadd.vv` sollicitent toutes deux les pipelines arithmétiques vectoriels du cœur X100, entrant ainsi en compétition pour les mêmes unités d'exécution. À l'inverse, les chargements vectoriels (`vle64.v`), pris en charge par des unités mémoire dédiées, s'exécutent efficacement en parallèle avec les opérations arithmétiques.

Ce constat nous a conduits à étendre MIPPV2 avec la fonction `mipp::fmaddi : rvd<T>` `mipp::fmaddi(rvd<T> acc, double scalar, rvd<T> vec)` Cette primitive tire parti de l'instruction native `vfadd.vf` de l'extension RVV. Elle réalise directement la multiplication-accumulation entre un vecteur et une valeur scalaire maintenue dans un registre flottant (Liste 14), éliminant ainsi toute instruction de diffusion intermédiaire :

```
1 for(size_t k = 0; k < A.cols; ++k) {
2     const auto a = A[i][k];
3     const auto b = load<T,lmul>(B[k] + j);
4
5     c = fmaddi(c, a, b);
6 }
```

C++

Liste 14 – Boucle d'accumulation optimisée utilisant `mipp::fmaddi` sans diffusion vectorielle.

À l'aide de cette primitive, un **micro-kernel** de la classe **Register Blocked** a été dimensionné en exploitant les spécificités de RVV :

1. **Paramétrage LMUL=4 et dimensionnement du bloc** : en fixant $LMUL = 4$, chaque vecteur logique rassemble 4 registres physiques (soit 1024 bits = 16 doubles). Les dimensions du bloc de registres sont fixées à $M_R = 3$ lignes et $N_R = 2 \times V_L = 32$ colonnes,
2. **Saturation intégrale du banc de registres** : le noyau alloue 6 vecteurs d'accumulateurs (3×2 , soit $6 \times 4 = 24$ registres physiques) et 2 vecteurs temporaires pour les chargements de B ($2 \times 4 = 8$ registres physiques). Les 32 registres physiques du processeur sont ainsi mobilisés simultanément sans déversement mémoire,
3. **Déroutage de boucle et pré-chargement** : la boucle k est déroulée d'un facteur 4. Les 12 scalaires de A nécessaires ($3 \text{ lignes} \times 4 \text{ pas}$) sont pré-chargés en début d'itération le long du panneau de A (Liste 23), optimisant l'ordonnancement entre chargements scalaires et calculs `vfadd.vf`. Des facteurs de déroulage supérieurs (6 et 8) ont été testés sans apporter de gain supplémentaire.

Ce **micro-kernel**, désigné par **X100-tuned MIPPV2 RB A-panel pack (GCC)**, traite un bloc de registres de $3 \times 32 = 96$ éléments en double précision (soit trois fois la capacité du bloc 4×8 de **Skylake**) et constitue l'implémentation la plus performante obtenue sur **X100**.

Une fois les schémas d'accumulation optimaux validés empiriquement sur **Skylake** et **X100**, nous avons cherché à les généraliser sous une forme logicielle plus concise. À partir de ces motifs, nous avons implémenté des versions génériques où la largeur du bloc de registres $N_R = V_L \times LMUL$ est paramétrée à la compilation (`template<int lmul>`). L'enjeu de cette démarche est de mesurer le compromis entre simplicité d'expression et efficacité : nous cherchons à déterminer si ces versions factorisées et paramétrables parviennent à approcher le débit des **micro-kernels** manuellement spécialisés pour chaque microarchitecture.

Ces variantes permettent de tester l'ensemble des facteurs de LMUL supportés par MIPPV2 en RVV (LMUL matériel) comme en AVX2 (LMUL émulé). Les mesures expérimentales (Fig. 16 et Fig. 17) confirment que les configurations LMUL = 2 et LMUL = 4 capturent l'essentiel des performances des versions sur-mesure, tandis que les valeurs extrêmes sous-performent :

- avec LMUL = 1, le volume d'instructions indépendantes en vol est insuffisant pour masquer la latence des unités de calcul arithmétique.
- avec LMUL = 8, la pression sur le banc de registres vectoriels devient excessive, provoquant des déversements mémoire (**spilling**) qui dégradent sévèrement le débit.

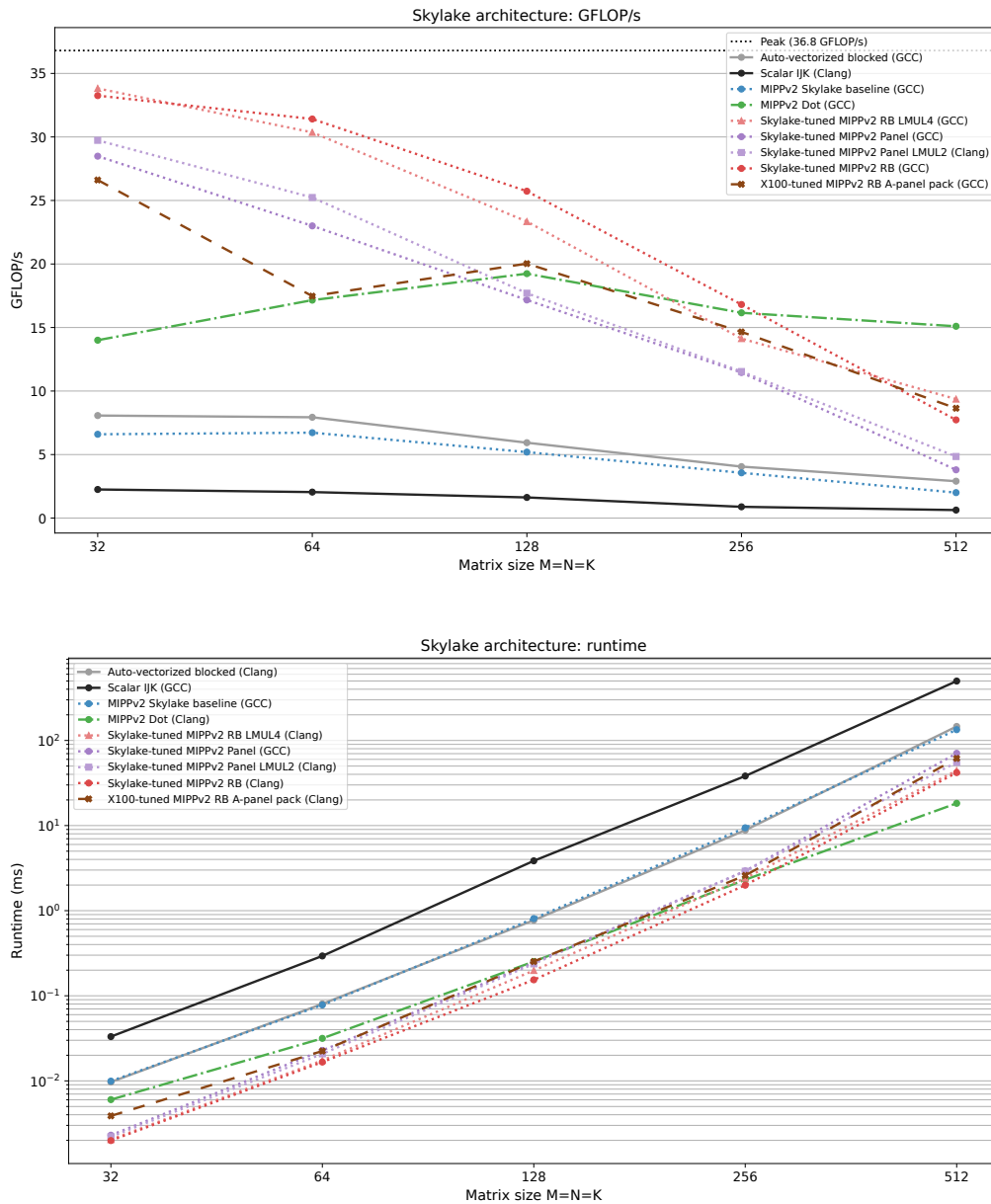


Fig. 16 – Temps d'exécution (en bas) et débit en GFLOP/s (en haut) de différents **micro-kernels** de GEMM sur Skylake.

La Fig. 16 présente l'évolution du temps d'exécution et du débit en GFLOP/s de différents **micro-kernels** pour des matrices d'entrée de dimensions 32×32 à 512×512 éléments sur Skylake (compilés avec l'option `-march=native`).

Les temps d'exécution et les débits sont obtenus en mesurant la durée totale d'un grand nombre d'itérations du **micro-kernel**, puis en divisant ce temps par le nombre d'itérations effectuées. Des itérations préalables de préchauffage (**warmup**) sont systématiquement réalisées pour garantir une exécution en cache chaud et éliminer les variations de latence mémoire. L'ensemble des mesures est conduit sur un seul cœur processeur.

L'analyse comparative des courbes met en évidence plusieurs régimes de performance :

- les versions scalaires (**Scalar ijk**) et vectorielles naïves (**MIPPV2 Skylake Baseline**) forment la borne basse de performance, pénalisées par l'absence de réutilisation des données et la dépendance stricte sur un accumulateur unique.
- la version auto-vectorisée par blocs (**Auto-Vectorized Block (GCC)**) adapte l'algorithme optimisé pour Skylake en utilisant des tableaux scalaires de 4 éléments à la place des registres vectoriels explicites. Cette structuration permet au compilateur de produire un code vectorisé plus efficace que la version MIPPV2 naïve, tout en restant inférieur aux versions MIPPV2 optimisées manuellement.
- la classe **Dot Product (MIPPV2 Dot (GCC))** offre un régime intermédiaire et présente la courbe la plus stable face à la montée en dimension des matrices, confirmant la régularité de son schéma d'accumulation.
- les courbes Skylake-tuned MIPPV2 RB (GCC) (classe **Register Blocked**) et Skylake-tuned MIPPV2 Panel (GCC) (classe **Panel Register Blocked**) correspondent aux versions des **micro-kernels** spécifiquement optimisées pour l'architecture Skylake. Elles atteignent les débits les plus élevés.
- les versions génériques paramétrées par LMUL (RB LMUL4 et Panel LMUL2) reprennent le même motif d'accumulation que les versions optimales pour Skylake en rendant la largeur du bloc paramétrable (M_R fixé, $N_R = V_L \times \text{LMUL}$). Cette approche permet d'explorer différentes configurations de blocs de registres sans modifier le **micro-kernel**. Elles maintiennent un niveau d'efficacité élevé mais restent légèrement inférieurs aux versions dédiées, quantifiant ainsi le compromis entre concision d'écriture et spécialisation manuelle poussée.
- enfin, le **micro-kernel** présentant les meilleures performances sur le X100 (X100-tuned RB A-panel pack) s'effondre sur Skylake. Cela met en évidence une mauvaise **portabilité des performances** de X100 vers Skylake. Ce **micro-kernel**, dimensionné pour exploiter 32 registres en RVV, il sature le banc de 16 registres de l'architecture AVX2 et provoque un déversement mémoire (**spilling**) massif.

Afin d'évaluer quantitativement l'efficacité des **micro-kernels** sur Skylake, nous comparons le débit maximal mesuré au débit crête théorique d'un cœur de cette architecture. Sur Skylake, un cœur dispose de deux unités d'exécution vectorielles dédiées aux instructions FMA (ports 0 et 1). Chaque instruction `vmaddpd` opère sur 4 valeurs en double précision et effectue 4 multiplications et 4 additions, soit 8 opérations flottantes. La capacité maximale de traitement par cycle est donc de : $2 \times 8 = 16$ FLOP/cycle. Le processeur étant cadencé à 2,3 GHz lors de nos expériences, le débit crête théorique sur un cœur s'établit à : $16 \times 2.3 = 36.8$ GFLOP/s. Le débit le plus élevé est atteint par Skylake-tuned MIPPV2 RB LMUL4 (Clang) avec 33.82 GFLOP/s pour des matrices de dimensions 32×32 , soit environ **92 %** du pic théorique de la machine.

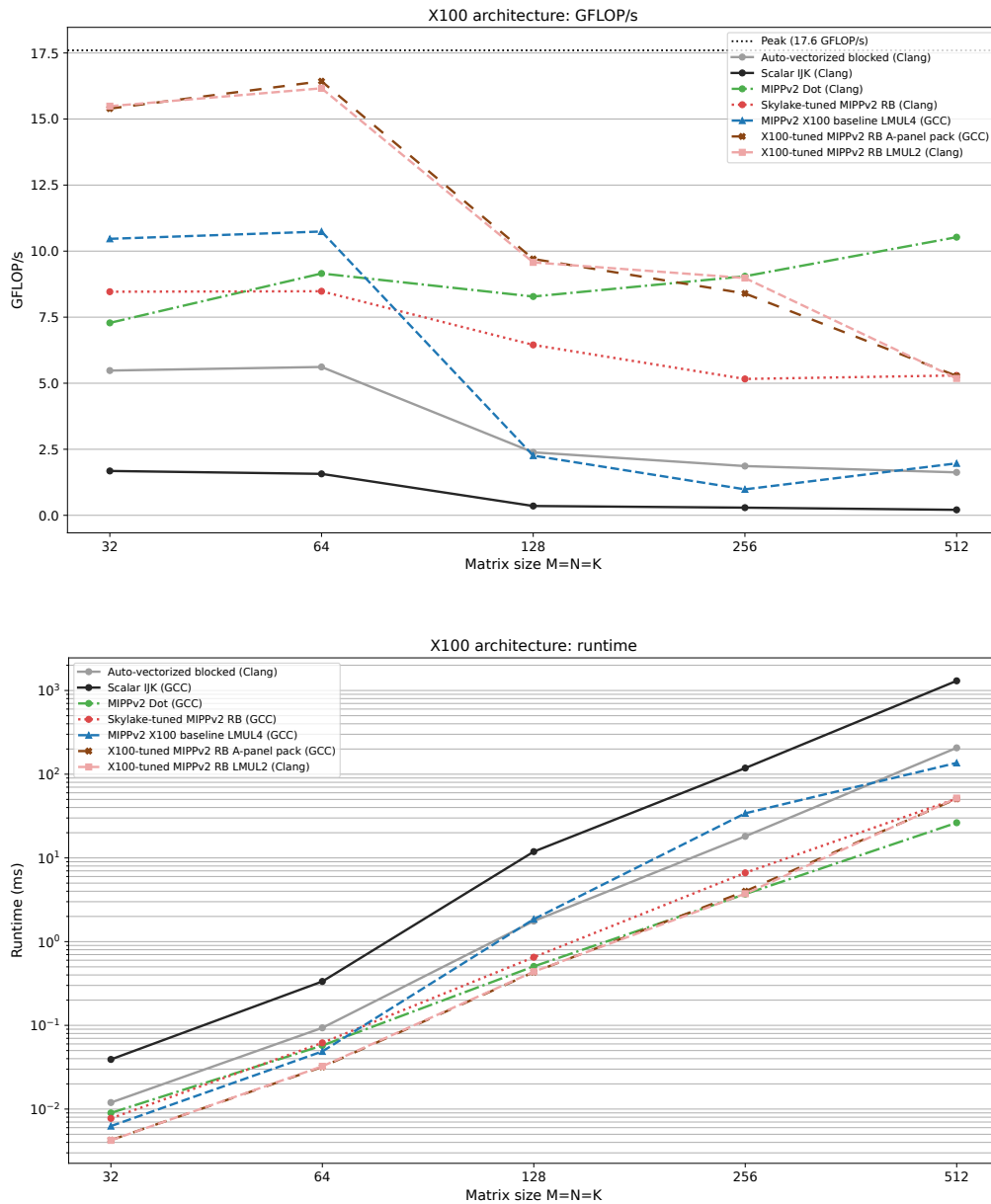


Fig. 17 – Temps d'exécution et débit en GFLOP/s de différents **micro-kernels** de GEMM sur X100.

La Fig. 17 compare le temps d'exécution et le débit en GFLOP/s en fonction de la dimension des matrices d'entrée pour les différents **micro-kernels** évalués sur le processeur X100.

- les versions scalaires (**Scalar IJK**) et auto-vectorisées (**Auto-vectorized blocked**) présentent les débits les plus faibles. La version auto-vectorisée, dont la structure en tableaux repose sur le dimensionnement 4×8 conçu pour **Skylake**, ne permet pas au compilateur d'exploiter efficacement les 32 registres de RVV ni d'éviter les instructions de diffusion, illustrant là encore la non-transposabilité directe d'un motif taillé pour une autre microarchitecture.
- la version **MIPV2 X100 baseline LMUL4** (GCC) exploite la primitive `fmaddi`. Cette implémentation correspond à une première vectorisation naïve. Elle permet de faire varier le LMUL. Les meilleures performances ayant été obtenues avec $LMUL = 4$, seule cette version est présentée. Elle surpasse nettement les versions non optimisées mais reste limitée par l'absence de réutilisation dans les registres.
- la classe **Dot Product** (**MIPV2 Dot**) reproduit un profil de performance très proche de celui obtenu sur **Skylake** : elle s'impose à nouveau comme l'approche la plus stable et la plus portable entre les deux architectures matérielles.
- le **micro-kernel** conçu pour **Skylake** (**Skylake-tuned MIPV2 RB**) sous-performe sur X100, obtenant des résultats inférieurs à la version **MIPV2 X100 baseline LMUL4** (GCC). Cette contre-performance s'explique à la fois par la contention créée par les diffusions `vfmv.v.f` sur les pipelines arithmétiques et par le sous-dimensionnement de son bloc (4×8), qui ne mobilise que 14 registres sur les 32 disponibles.
- les versions optimisées pour X100 dominent nettement l'ensemble du benchmark grâce à l'élimination des diffusions via `fmaddi` et à un blocage de registres adapté : la version spécialisée **X100-tuned RB A-panel pack** tire parti de son bloc maximal de 3×32 éléments ($LMUL = 4$), tandis que la variante paramétrée LMUL2 confirme l'efficacité d'un dimensionnement intermédiaire.

Comme pour **Skylake**, nous évaluons quantitativement l'efficacité du meilleur noyau par rapport au débit crête théorique du cœur X100. Le cœur dispose de deux pipelines arithmétiques vectoriels de 128 bits. Chaque pipeline traitant 2 doubles par cycle avec une opération `vfmadd` (2 opérations flottantes par élément), la capacité maximale de traitement est de : $2 \times (2 \times 2) = 8$ FLOP/cycle. Le processeur étant cadencé à 2,2 GHz lors de nos expériences, le débit crête théorique sur un cœur s'établit à : $8 \times 2.2 = 17.6$ GFLOP/s. Le débit le plus élevé est atteint par **X100-tuned MIPV2 RB A-panel pack** (GCC) avec 16.43 GFLOP/s pour des matrices de dimensions 64×64 , soit environ **93 %** du pic théorique de la machine.

Ce cas d'étude sur le produit matriciel se distingue des évaluations précédentes : l'objectif n'est plus seulement de mesurer l'absence de surcoût ou de surpasser l'auto-vectorisation, mais de déterminer si **MIPV2** permet d'atteindre les limites calculatoires

physiques du matériel ($> 90\%$ du pic théorique) tout en analysant la portabilité des performances entre deux architectures aux paradigmes vectoriels distincts (AVX2 et RVV).

Cette étude permet de dégager trois conclusions principales :

1. **l'atteinte d'une efficacité calculatoire proche de l'optimum** : les meilleures implémentations MIPPV2 sur Skylake et X100 atteignent respectivement 92% et 93% du débit crête théorique matériel. Ce résultat est comparable aux performances des **micro-kernels** des bibliothèques BLAS et est particulièrement remarquable sur X100, une architecture récente et moins étudiée que Skylake,
2. **l'accélération de l'exploration de l'espace de conception** : l'abstraction du LMUL introduite dans MIPPV2 permet de balayer rapidement différentes configurations de blocs de registres et de facteurs de déroulage par simple instanciation de gabarits statiques, réduisant drastiquement l'effort de développement face à une réécriture manuelle ,
3. **la distinction entre optimisations paramétriques et structurelles** : si l'ajustement des dimensions d'un noyau (taille du bloc de registre, LMUL, déroulage) est facilement explorable à l'aide de MIPP, le choix du motif d'accumulation lui-même (comme la transition d'un schéma `broadcast + fmadd` vers un schéma `fmaddi`) reste dicté par la microarchitecture et l'ISA. Une bibliothèque SIMD ne peut donc éliminer à elle seule le besoin d'adapter le schéma algorithmique pour garantir la portabilité des performances entre cibles hétérogènes.

4.0.4 | Conclusion des mesures de performance L'ensemble des campagnes expérimentales menées à travers ces trois cas d'études permet de dégager les conclusions suivantes :

1. **la portabilité** : MIPPV2 permet d'écrire du code portable et efficace,
2. **l'absence de surcoût d'abstraction** : les performances obtenues avec MIPPV2 sont comparables à celles d'une implémentation reposant directement sur des intrinsèques, ce qui confirme que MIPPV2 n'induit pas de surcoût significatif dans les cas étudiés,
3. **la non-régression** : aucune régression de performance notable n'a été observée entre MIPPV1 et MIPPV2,
4. **la démarche empirique face à l'auto-vectorisation** : pour certaines classes de *kernels*, l'auto-vectorisation du compilateur constitue généralement une solution suffisante n'appelant pas nécessairement d'optimisation manuelle. Le recours à MIPPV2 (ou à une vectorisation explicite) est surtout pertinent lorsque l'on a vérifié **expérimentalement** que le compilateur ne parvient pas à générer un code vectoriel performant,
5. **les cas d'usage privilégiés de la vectorisation explicite** : l'utilisation de MIPPV2 s'avère particulièrement pertinente pour des algorithmes nécessitant des restructurations spécifiques (masquage **branch-free**, chemins rapides **fastpaths**)

ou pour des calculs faisant appel à des fonctions mathématiques avancées comme les transcendantes vectorisées,

6. **la capacité à saturer les unités de calcul** : l'évaluation sur le **micro-kernel** de GEMM prouve que l'abstraction de la bibliothèque permet d'atteindre le régime calculatoire crête du processeur, délivrant plus de 90 % du débit maximal théorique aussi bien sur Skylake que sur X100,
7. **le gain de productivité dans l'exploration de l'espace de conception** : des abstractions comme le LMUL permettent d'explorer rapidement différentes tailles de blocs de registres et facteurs de déroulage par instanciation de gabarits statiques, tout en mettant en lumière la frontière entre optimisations paramétriques (portables) et choix de motifs d'accumulation (liés à la microarchitecture).

Il serait toutefois nécessaire d'étendre ces benchmarks aux architectures ARM NEON et ARM SVE, ainsi que d'évaluer un ensemble plus large de *kernels* et de cas d'utilisation, afin de déterminer dans quelle mesure ces conclusions peuvent être généralisées. Les résultats obtenus restent néanmoins encourageants quant à la pertinence de l'approche proposée par MIPPv2.

5 | Conclusion, Perspectives & Ouvertures

5.1 | Conclusion

Ce stage s'est articulé autour de deux objectifs fondamentaux : faire progresser significativement la refonte de MIPPv2 en hissant le support de l'extension RISC-V (RVV) au même niveau de maturité que les autres architectures supportées par MIPP, et valider rigoureusement l'absence de surcoût d'abstraction de la bibliothèque en conditions industrielles et applicatives avancées.

Les résultats obtenus à travers ce travail se déclinent selon trois contributions majeures :

1. **l'unification logicielle et l'intégration native de RVV** : l'infrastructure de génération de code basée sur Python et Jinja2 a été enrichie afin de produire. Ce générateur intègre désormais le support complet de RVV 1.0 (masquage/prédication native, groupement de registres LMUL matériel et émulé), tout en élargissant et améliorant le support des jeux d'instructions SSE, AVX2, AVX-512 et NEON. L'ensemble est soutenu par une suite automatisée de tests unitaires et de génération de documentation,
2. **la validation industrielle et la caractérisation des cas d'usage** : la collaboration avec l'IFPEN a permis de développer la couche d'interopérabilité MIPPv2.h, assurant la compatibilité avec le solveur linéaire industriel MCGSolver. Les mesures conduites confirment l'absence de surcoût face aux intrinsèques et l'absence de régression vis-à-vis de MIPPv1. Ces évaluations permettent également d'identifier

clairement les classes de **kernels** où la vectorisation explicite avec MIPPV2 apporte un gain mesurable par rapport à l'auto-vectorisation, offrant ainsi aux développeurs un guide précieux pour déterminer quand son utilisation est pertinente,

3. la **saturation matérielle et l'exploration de l'espace de conception** : l'optimisation du **micro-kernel** de GEMM démontre que les abstractions de MIPPV2 permettent de concevoir des noyaux atteignant le débit crête des processeurs, dépassant 90 % du pic théorique sur Skylake comme sur SpacemiT X100. Ces travaux illustrent l'intérêt de l'abstraction LMUL pour accélérer l'exploration systématique de l'espace d'optimisation (taille des blocs de registres, déroulage), tout en mettant en évidence les frontières de la portabilité des performances entre microarchitectures hétérogènes.

En combinant portabilité multi-architectures, absence de surcoût d'abstraction et clarté sur ses domaines d'application, MIPPV2 pose des bases solides pour son intégration dans des codes scientifiques en production et pour sa future mise à disposition en **open-source**.

5.2 | Perspectives

Sur le plan du développement logiciel, plusieurs étapes restent nécessaires avant d'envisager la mise à disposition de MIPPV2 en **open-source**. Il conviendra d'abord d'assurer une rétrocompatibilité complète avec les codes existants basés sur MIPPV1 en réintégrant certaines primitives manquantes, notamment les opérations de réorganisation vectorielle (`mipp::shuffle`, `mipp::interleave`) et des fonctions transcendentes spécifiques. Par ailleurs, la couche objet de MIPPV2, actuellement au stade de prototype, devra être finalisée afin d'offrir une interface expressive reposant sur la surcharge d'opérateurs tout en préservant le principe d'abstraction à coût nul.

Les cas d'études applicatifs menés durant ce stage ont également mis en lumière des pistes d'extension majeures pour le générateur de code. Il serait particulièrement pertinent d'y intégrer de manière pérenne le support des opérations mixtes vecteur-scalaire (à l'image de `mipp::fmaddi`), indispensables pour éviter les diffusions redondantes sur des architectures comme RVV. De même, l'introduction de types **Tuples** enrichis permettrait d'automatiser l'allocation de blocs de registres ($M_R \times N_R$) et le déroulage multi-vectorel pour les noyaux denses. L'ajout d'abstractions de méta-programmation statique faciliterait également l'exploration systématique de l'espace de conception directement depuis le générateur.

Enfin, l'évaluation expérimentale de la bibliothèque gagnerait à être approfondie sur le plan de l'efficacité énergétique. L'utilisation d'outils dédiés tels que `likwid` ou `NCM` permettrait de mesurer la consommation électrique des différentes implémentations et de caractériser le compromis performance-énergie (**Energy-Delay Product**) des optimisations manuelles par rapport à l'auto-vectorisation. Ces mesures devront s'accompagner d'un suivi de la maturation des compilateurs pour RVV et d'une extension

des benchmarks à de nouvelles cibles matérielles, en particulier ARM SVE à longueur de vecteur variable.

5.3 | Ouvertures

L'évolution à long terme de MIPPV2 ouvre plusieurs perspectives prometteuses à l'intersection de l'ingénierie des compilateurs, du calcul haute performance et de l'intelligence artificielle :

Une première piste consisterait à faire évoluer le générateur vers un système d'**auto-tuning** adaptatif. En s'appuyant sur une base exhaustive de micro-benchmarks caractérisant chaque primitive sur les différentes cibles matérielles, le générateur pourrait sélectionner automatiquement les implémentations les plus performantes par défaut. À l'échelle applicative, un tel moteur permettrait d'explorer systématiquement les combinaisons d'options (variantes de fonctions, facteurs de déroulage, tuilage) afin d'optimiser un noyau selon des critères précis (débit crête, latence minimale ou compromis énergétique).

Une seconde perspective réside dans la spécialisation de MIPP pour l'inférence et le traitement de données en basse précision. L'intégration de types de données compacts et quantifiés (`int8`, `int4`, `bfloat16`, `float16`), couplée à des primitives d'opérations sur de petites matrices, permettrait d'exploiter efficacement les accélérateurs vectoriels et matriciels récents (tels qu'Intel AMX ou les extensions matricielles de RISC-V). La philosophie de MIPP — exploiter le matériel lorsqu'il est disponible et émuler les opérations dans le cas contraire — s'appliquerait naturellement à ces nouveaux formats.

6 | Annexes

Les tableaux suivant sont des reproductions de la documentation des intrinsèques RVV[9].

Tableau 2 – Tableau des types définis par `riscv_vector.h` pour $EMUL \geq 1$.

Types	EMUL=1	EMUL=2	EMUL=4	EMUL=8
<code>int8_t</code>	<code>vint8m1_t</code>	<code>vint8m2_t</code>	<code>vint8m4_t</code>	<code>vint8m8_t</code>
<code>int16_t</code>	<code>vint16m1_t</code>	<code>vint16m2_t</code>	<code>vint16m4_t</code>	<code>vint16m8_t</code>
<code>int32_t</code>	<code>vint32m1_t</code>	<code>vint32m2_t</code>	<code>vint32m4_t</code>	<code>vint32m8_t</code>
<code>int64_t</code>	<code>vint64m1_t*</code>	<code>vint64m2_t*</code>	<code>vint64m4_t*</code>	<code>vint64m8_t*</code>
<code>uint8_t</code>	<code>vuint8m1_t</code>	<code>vuint8m2_t</code>	<code>vuint8m4_t</code>	<code>vuint8m8_t</code>
<code>uint16_t</code>	<code>vuint16m1_t</code>	<code>vuint16m2_t</code>	<code>vuint16m4_t</code>	<code>vuint16m8_t</code>
<code>uint32_t</code>	<code>vuint32m1_t</code>	<code>vuint32m2_t</code>	<code>vuint32m4_t</code>	<code>vuint32m8_t</code>
<code>uint64_t</code>	<code>vuint64m1_t*</code>	<code>vuint64m2_t*</code>	<code>vuint64m4_t*</code>	<code>vuint64m8_t*</code>
<code>_Float16</code>	<code>vfloat16m1_t</code>	<code>vfloat16m2_t</code>	<code>vfloat16m4_t</code>	<code>vfloat16m8_t</code>
<code>float</code>	<code>vfloat32m1_t</code>	<code>vfloat32m2_t</code>	<code>vfloat32m4_t</code>	<code>vfloat32m8_t</code>
<code>double</code>	<code>vfloat64m1_t</code>	<code>vfloat64m2_t</code>	<code>vfloat64m4_t</code>	<code>vfloat64m8_t</code>

Tableau 3 – Tableau des types définis par `riscv_vector.h` pour $EMUL < 1$.

Types	EMUL= $\frac{1}{2}$	EMUL= $\frac{1}{4}$	EMUL= $\frac{1}{8}$
<code>int8_t</code>	<code>vint8mf2_t</code>	<code>vint8mf4_t</code>	<code>vint8mf8_t*</code>
<code>int16_t</code>	<code>vint16mf2_t</code>	<code>vint16mf4_t*</code>	N/A
<code>int32_t</code>	<code>vint32mf2_t*</code>	N/A	N/A
<code>int64_t</code>	N/A	N/A	N/A
<code>uint8_t</code>	<code>vuint8mf2_t</code>	<code>vuint8mf4_t</code>	<code>vuint8mf8_t*</code>
<code>uint16_t</code>	<code>vuint16mf2_t</code>	<code>vuint16mf4_t*</code>	N/A
<code>uint32_t</code>	<code>vuint32mf2_t*</code>	N/A	N/A
<code>uint64_t</code>	N/A	N/A	N/A
<code>_Float16</code>	<code>vfloat16mf2_t</code>	<code>vfloat16mf4_t*</code>	N/A
<code>float</code>	<code>vfloat32mf2_t*</code>	N/A	N/A
<code>double</code>	N/A	N/A	N/A

Tableau 4 – Tableau des types définis par `riscv_vector.h` pour $EMUL < 1$.

Types	n=1	n=2	n=4	n=8	n=16	n=32	n=64
<code>bool</code>	<code>vbool1_t</code>	<code>vbool2_t</code>	<code>vbool4_t</code>	<code>vbool8_t</code>	<code>vbool16_t</code>	<code>vbool32_t</code>	<code>vbool64_t*</code>

n.b : Les types suivis d'une astérisque indiquent qu'ils ne sont définis que si $ELEN \geq 64$. Les types associés aux types scalaires `_Float16`, `float` et `double` ne sont définis que si les extensions flottantes appropriées sont définies dans l'architecture (les extensions `zvf` et `zvfmin` pour `_Float16`, `zve32f` pour `float` et `zve32d` pour `double`).

Tableau 5 – Comparatif des fonctionnalités matérielles selon les jeux d'instructions supportés par MIPP.

	SSE	AVX	AVX-512	NEON	SVE	RVV
Taille des registres (bits)	128	256	512	128	Variable	Variable
Fonctions prédiquées	Non	Non	Oui	Non	Oui	Oui
LMUL	Non	Non	Non	Non	Non	Oui (1 à 8)
LDIV	Non	Oui ($\frac{1}{2}$)	Oui ($\frac{1}{2}, \frac{1}{4}$)	Non	Non	Dépend du type et de l'architecture ($\frac{1}{2}$ à $\frac{1}{8}$)

```

1 static inline rvd_float64_t mipp_fmadd_float64(
2     const rvd_float64_t r0,
3     const rvd_float64_t r1,
4     const rvd_float64_t r2) {
5     #if defined(MIPP_AVX512)
6         return mipp_avx512_fmadd_float64(r0, r1, r2);
7     #elif defined(MIPP_AVX)
8         return mipp_avx_fmadd_float64(r0, r1, r2);
9     #elif defined(MIPP_SSE)
10        return mipp_sse_fmadd_float64(r0, r1, r2);
11    #elif defined(MIPP_RVV)
12        return mipp_rvv_fmadd_float64(r0, r1, r2);
13    #elif defined(MIPP_NEON)
14        return mipp_neon_fmadd_float64(r0, r1, r2);
15    #elif defined(MIPP_SVE)
16        return mipp_sve_fmadd_float64(r0, r1, r2);
17    #else
18        printf("MIPP panic: unsupported case, this should never happen.\n");
19        exit(-1);
20    #endif
21 }
```

h C Header

Liste 15 – Renommage de la fonction fmadd.

```

1 // code MIPP généré
2 inline rvd<float64_t,AVX> fmadd(const rvd<float64_t,AVX> r0,
3                               const rvd<float64_t,AVX> r1,
4                               const rvd<float64_t,AVX> r2) {
5     return mipp_avx_fmadd_float64(r0.r, r1.r, r2.r)
6 }
7 inline rvd<float64_t,SSE> fmadd(const rvd<float64_t,SSE> r0,
8                               const rvd<float64_t,SSE> r1,
9                               const rvd<float64_t,SSE> r2) {
10    return mipp_sse_fmadd_float64(r0.r, r1.r, r2.r)
11 }

```

Liste 16 – Exemples de l'utilisation de la surcharge de fonction pour définir un `fmadd` générique entre AVX et SSE.

```

1  inline void eval_naive_axpy(const FpComp          alpha,
2                             const std::vector<FpComp> &a,
3                             std::vector<FpComp>      &b) const
4  {
5      for(size_t i = 0; i < a.size(); ++i)
6          b[i] += alpha * a[i];
7  }
8
9  inline void eval_naive_axpydots(const std::vector<FpComp> &a,
10                                 const std::vector<FpComp> &b,
11                                 const std::vector<FpComp> &c,
12                                 const FpComp          alpha,
13                                 const FpComp          omega,
14
15                                 std::vector<FpComp> &d,
16                                 std::vector<FpComp> &e,
17                                 FpStore            &dot,
18                                 FpStore            &sqrnm2) const
19  {
20      dot = 0;
21      sqrnm2 = 0;
22      for(size_t i = 0; i < a.size(); ++i)
23      {
24          d[i] = d[i] + omega * e[i] + alpha * a[i];
25          e[i] = e[i] - omega * b[i];
26
27          sqrnm2 += e[i] * e[i];
28          dot += e[i] * c[i];
29      }
30  }
31  inline void eval_naive_dot(const std::vector<FpComp> &a,
32                             const std::vector<FpComp> &b,
33                             FpStore            &dot) const
34  {
35      dot = 0;
36      for(size_t i = 0; i < a.size(); ++i)
37          dot += a[i] * b[i];
38  }
39
40  inline void eval_naive_dotsqrm2(const std::vector<FpComp> &a,
41                                  const std::vector<FpComp> &b,
42                                  FpStore            &dot,
43                                  FpStore            &sqrnm2) const
44  {
45      dot = 0;
46      sqrnm2 = 0;
47      for(size_t i = 0; i < a.size(); ++i)
48      {
49          sqrnm2 += a[i] * a[i];
50          dot += a[i] * b[i];
51      }
52  }

```

```

1  inline void eval_naive_scal(const FpComp          alpha,
2                             std::vector<FpComp> &a) const
3  {
4      for(size_t i = 0; i < a.size(); ++i)
5          a[i] *= alpha;
6  }
7  inline void eval_naive_sqnorm2(const std::vector<FpComp> &a,
8                                 FpStore                  &norm) const
9  {
10     norm = 0;
11     for(size_t i = 0; i < a.size(); ++i)
12         norm += a[i] * a[i];
13 }

```

Liste 17 – Version scalaire des différents *kernels* utilisés dans les *benchmarks* unitaires d'algèbre linéaire.

```

1  inline void eval_naive_mol_mass(const std::vector<FpComp> &X,
2                                 std::vector<FpStore> &Mphi,
3                                 std::vector<FpStore> &dMphi_dX) const
4  {
5      const size_t num_species = m_M.size();
6      const size_t num_cells = X.size() / num_species;
7      const size_t full_elements = num_cells * num_species;
8
9      for (size_t c = 0; c < num_cells; ++c) {
10         FpComp cell_molar_mass = 0;
11         for (size_t k = 0; k < num_species; ++k) {
12             size_t idx = c * num_species + k;
13             FpComp M_k = m_M[k];
14             cell_molar_mass += X[idx] * M_k;
15             dMphi_dX[idx] = static_cast<FpStore>(M_k);
16         }
17         Mphi[c] = static_cast<FpStore>(cell_molar_mass);
18     }
19
20     if (full_elements < X.size()) {
21         FpComp tail_molar_mass = 0;
22         for (size_t i = full_elements; i < X.size(); ++i) {
23             size_t k = i % num_species;
24             FpComp M_k = m_M[k];
25             tail_molar_mass += X[i] * M_k;
26             dMphi_dX[i] = static_cast<FpStore>(M_k);
27         }
28         Mphi[num_cells] = static_cast<FpStore>(tail_molar_mass);
29     }
30 }

```

```

1
2  inline void eval_naive_cppl(const std::vector<FpComp> &S,
3                             std::vector<FpStore> &Pc,
4                             std::vector<FpStore> &dPc_dS) const
5  {
6      for (size_t i = 0; i < S.size(); ++i) {
7          eval_scalar_naive(S[i], Pc[i], dPc_dS[i]);
8      }
9  }
10
11 inline void eval_scalar_naive(const FpComp S,
12                              FpStore &Pc,
13                              FpStore &dPc_dS) const
14 {
15     FpComp dSe_dS;
16     const FpComp Se = computeSe_naive(S, dSe_dS);
17     FpComp alpha = (FpComp)-1 / m_lambda;
18     if (Se) {
19         Pc = m_Pe * pow(Se, alpha);
20         dPc_dS = alpha * m_Pe * dSe_dS * pow(Se, alpha - 1);
21     } else {
22         Pc = 0.; // infinity
23         dPc_dS = 0.; // -infinity
24     }
25 }
26 inline void eval_naive_rppl(const std::vector<FpComp> &Sw,
27                             std::vector<FpStore> &krw,
28                             std::vector<FpStore> &dkrw_dSw) const {
29     for(size_t i = 0; i < Sw.size(); ++i)
30     {
31         FpComp dSwBar_dSw;
32         const FpComp SwBar = computeSwBar_naive(Sw[i], dSwBar_dSw);
33         krw[i] = pow(SwBar, m_alpha);
34         dkrw_dSw[i] = dSwBar_dSw * m_alpha * pow(SwBar, m_alpha - 1);
35     }
36 }
37
38 inline FpComp computeSwBar_naive(const FpComp Sw,
39                                 FpComp &dSwBar_dSw) const {
40     FpComp SwBar;
41     if (Sw < m_Swi) {
42         SwBar = 0;
43         dSwBar_dSw = 0;
44     } else if (Sw <= 1 - m_Sgc) {
45         SwBar = (Sw - m_Swi) / (1 - m_Sgc - m_Swi);
46         dSwBar_dSw = 1 / (1 - m_Sgc - m_Swi);
47     } else {
48         SwBar = 1;
49         dSwBar_dSw = 0;
50     }
51     return SwBar;
52 }

```

Liste 18 – Version scalaire des *kernels* masse molaire, pression capillaire et perméabilité relative utilisés dans les benchmarks unitaires de physique.

```

1  inline void eval_naive_peng_rob(const std::vector<FpComp> &A,
2                                const std::vector<FpComp> &B,
3                                std::vector<FpStore> &Z,
4                                std::vector<FpStore> &dZ_dC0,
5                                std::vector<FpStore> &dZ_dC1,
6                                std::vector<FpStore> &dZ_dC2)
7  {
8      for (size_t i = 0; i < A.size(); ++i) {
9          eval_scalar_naive(A[i], B[i], Z[i], dZ_dC0[i], dZ_dC1[i], dZ_dC2[i]);
10     }
11 }
12 inline void eval_scalar_naive(const FpComp A,
13                               const FpComp B,
14                               FpStore &Z,
15                               FpStore &dZ_dC0,
16                               FpStore &dZ_dC1,
17                               FpStore &dZ_dC2)
18 {
19     FpComp C0 = 0., C1 = 0., C2 = 0.;
20     computePolynomialCoeff(A, B, C0, C1, C2);
21     fZ(C0, C1, C2, A, B, Z, dZ_dC0, dZ_dC1, dZ_dC2);
22 }
23
24 inline FpComp fSelectZ(FpComp &X1,
25                        FpComp &X2,
26                        FpComp &X3,
27                        bool &isX2I,
28                        const FpComp A,
29                        const FpComp B) const
30 {
31     FpComp Z = 0;
32
33     if (isX2I == true) Z = X1;
34     else {
35         FpComp Zmin = std::min({X1, X2, X3});
36         FpComp Zmax = std::max({X1, X2, X3});
37
38         if (Zmin <= B) Z = Zmax;
39         else {
40             FpComp G_Zmin = fG(Zmin, A, B);
41             FpComp G_Zmax = fG(Zmax, A, B);
42
43             if (G_Zmin < G_Zmax) {
44                 Z = Zmin;
45             } else {
46                 Z = Zmax;
47             }
48         }
49     }
50     return Z;
51 }

```

C++

```

1  inline void fSolveCubic(const FpComp B,
2                          const FpComp C,
3                          const FpComp D,
4                          FpComp &X1,
5                          FpComp &X2,
6                          FpComp &X3,
7                          bool &isX2I) const
8  {
9      X1 = 0; X2 = 0; X3 = 0;
10     isX2I = false;
11
12     FpComp pi = (FpComp)3.141592653589793;
13     FpComp P, Q, FF, R, PHI3;
14
15     if ((C == 0) && (D == 0)) {
16         X1 = -B; X2 = 0; X3 = 0;
17         isX2I = false;
18     } else {
19         Q = (B * B * B * (FpComp)2 / (FpComp)27.0 - B * C / (FpComp)3.0 + D) / (FpComp)2.0;
20         P = (C * (FpComp)3 - B * B) / (FpComp)9.0;
21
22         FF = std::fabs(P);
23         R = std::sqrt(FF);
24         FF = R * Q;
25
26         if (FF < 0) R = -R;
27         FF = Q / (R * R * R);
28
29         if (P > 0) {
30             PHI3 = std::log(FF + std::sqrt(FF * FF + (FpComp)1)) / (FpComp)3.0;
31             X1 = -R * (std::exp(PHI3) - std::exp(-PHI3)) - B / (FpComp)3.0;
32             isX2I = true;
33         } else {
34             if (Q * Q + P * P * P > 0) {
35                 PHI3 = std::log(FF + std::sqrt(FF * FF - (FpComp)1.0)) / (FpComp)3.0;
36                 X1 = -R * (std::exp(PHI3) + std::exp(-PHI3)) - B / (FpComp)3.0;
37                 isX2I = true;
38             } else {
39                 PHI3 = std::atan(std::sqrt((FpComp)1.0 - FF * FF) / FF) / (FpComp)3.0;
40                 X1 = -R * std::cos(PHI3) * (FpComp)2 - B / (FpComp)3.0;
41                 X2 = R * std::cos(pi / (FpComp)3.0 - PHI3) * (FpComp)2 - B / (FpComp)3.0;
42                 X3 = R * std::cos(pi / (FpComp)3.0 + PHI3) * (FpComp)2 - B / (FpComp)3.0;
43                 isX2I = false;
44             }
45         }
46     }
47 }

```

Liste 19 – Extrait de la version scalaire du *kernel* Peng-Robinson utilisé dans les benchmarks unitaires de physique.

```

1  for(int iter = 0; iter < nb_iter; iter++)
2  {
3      algebra.copy(p, u);
4      alpha = algebra.dot(u, r0);
5      algebra.y_axpy(-alpha, u, r);
6      algebra.copy(r, v);
7
8      red_val = algebra.dotSqrnm2(r, v);
9
10     omega = red_val.first;
11     beta = red_val.second;
12
13     red_val = algebra.X_R_Res_Rho1(r0, v, p, omega, beta, alpha, x, r);
14
15     algebra.P(r, u, beta, omega, p);
16
17     rho1 = red_val.second;
18     beta = rho1/rho*alpha/omega;
19     rho = rho1;
20
21     res_init = algebra.IterSet(b, r0, p, x, r);
22 } // end of loop
23
24 result = algebra.sqrnm2(p);

```

Liste 20 – Boucle principale du *benchmark* pseudo-solveur.

Les figures suivantes (Fig. 18, Fig. 19, Fig. 20, Fig. 21) présentent des résultats complémentaires aux *benchmarks* unitaires d'algèbre linéaire et de physique, évaluant l'impact de la dimension des vecteurs et de la précision flottante (`float` et `double`). Des mesures similaires ont également été conduites sur *Skylake* ; les tendances observées étant identiques, seuls les résultats obtenus sur *Zen4* sont rapportés ici.

Ces graphiques confirment que les *kernels* MIPpv2 surpassant l'auto-vectorisation conservent systématiquement leur avantage, quels que soient la taille des vecteurs ou le format flottant considéré. La seule exception concerne le *kernel* de masse molaire : sur de grands vecteurs en simple précision (`float`), la version auto-vectorisée par le compilateur devient plus performante que MIPpv2, alors que cette dernière l'emporte dans tous les autres cas de figure.

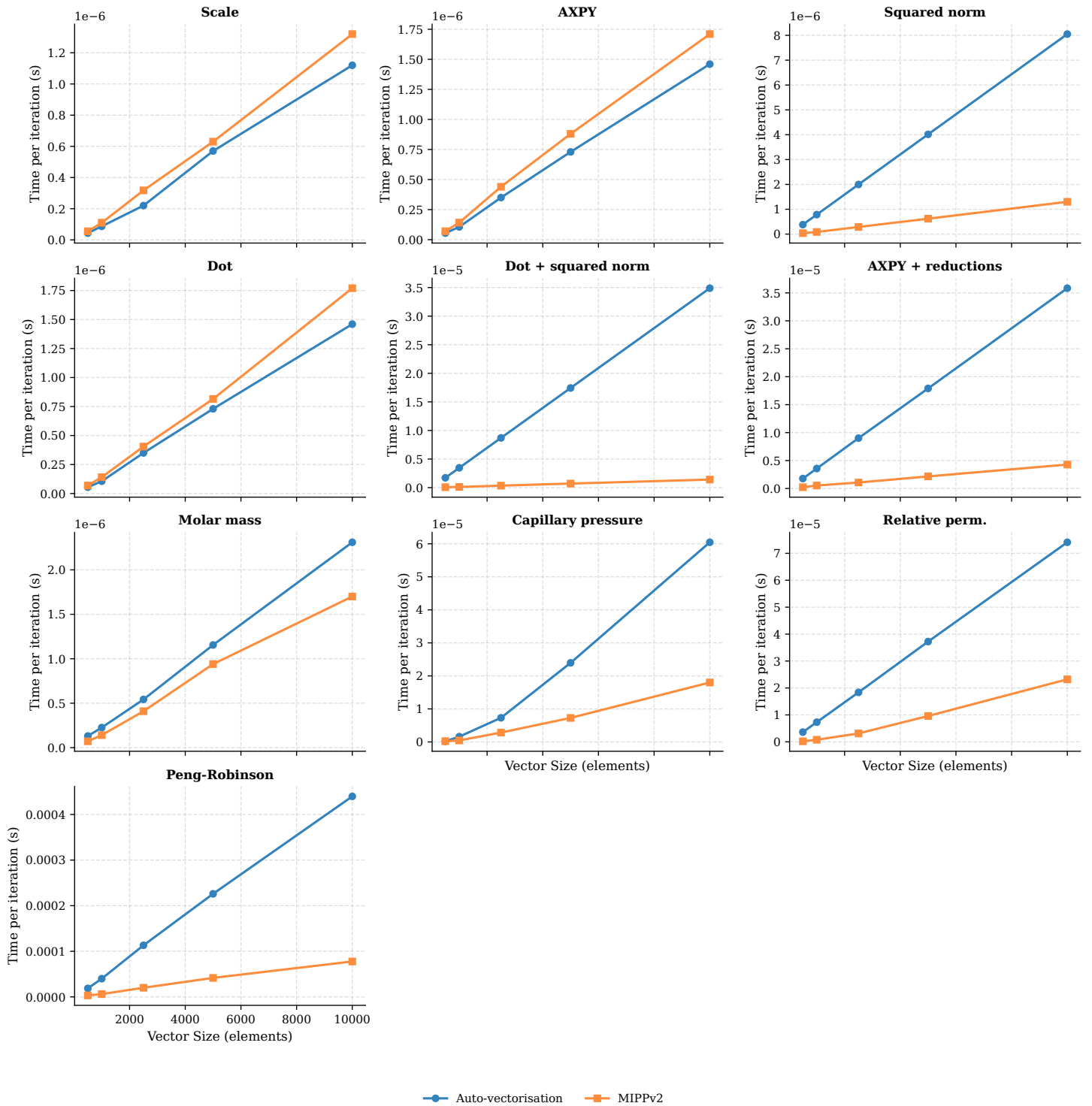


Fig. 18 – Temps d'exécution en millisecondes d'une itération de chaque *kernel* en fonction de la taille des vecteurs d'entrée pour des flottants **double** précision sur Zen4.

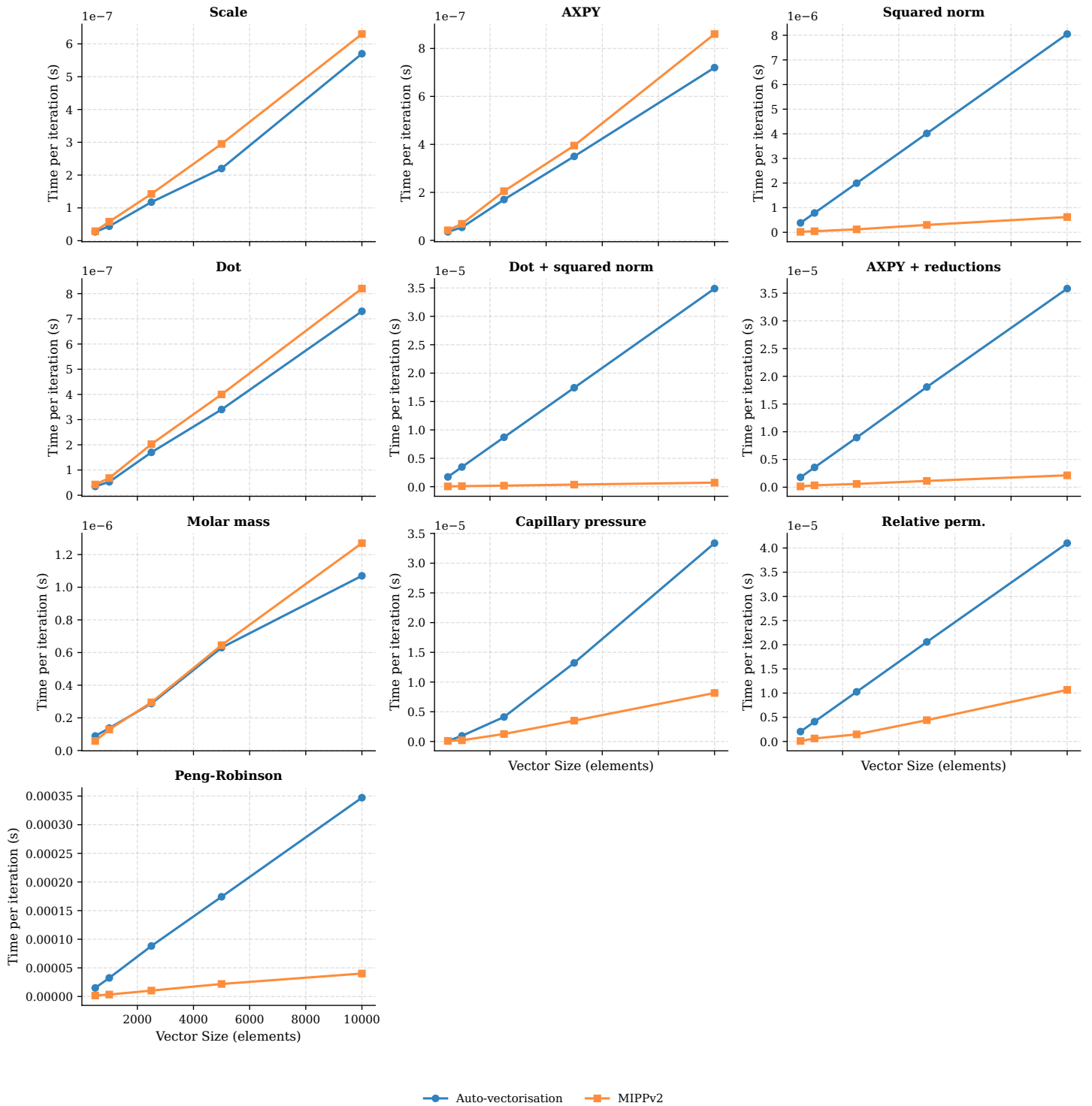


Fig. 19 – Temps d'exécution en millisecondes d'une itération de chaque *kernel* en fonction de la taille des vecteurs d'entrée pour des flottants **simple** précision sur Zen4.

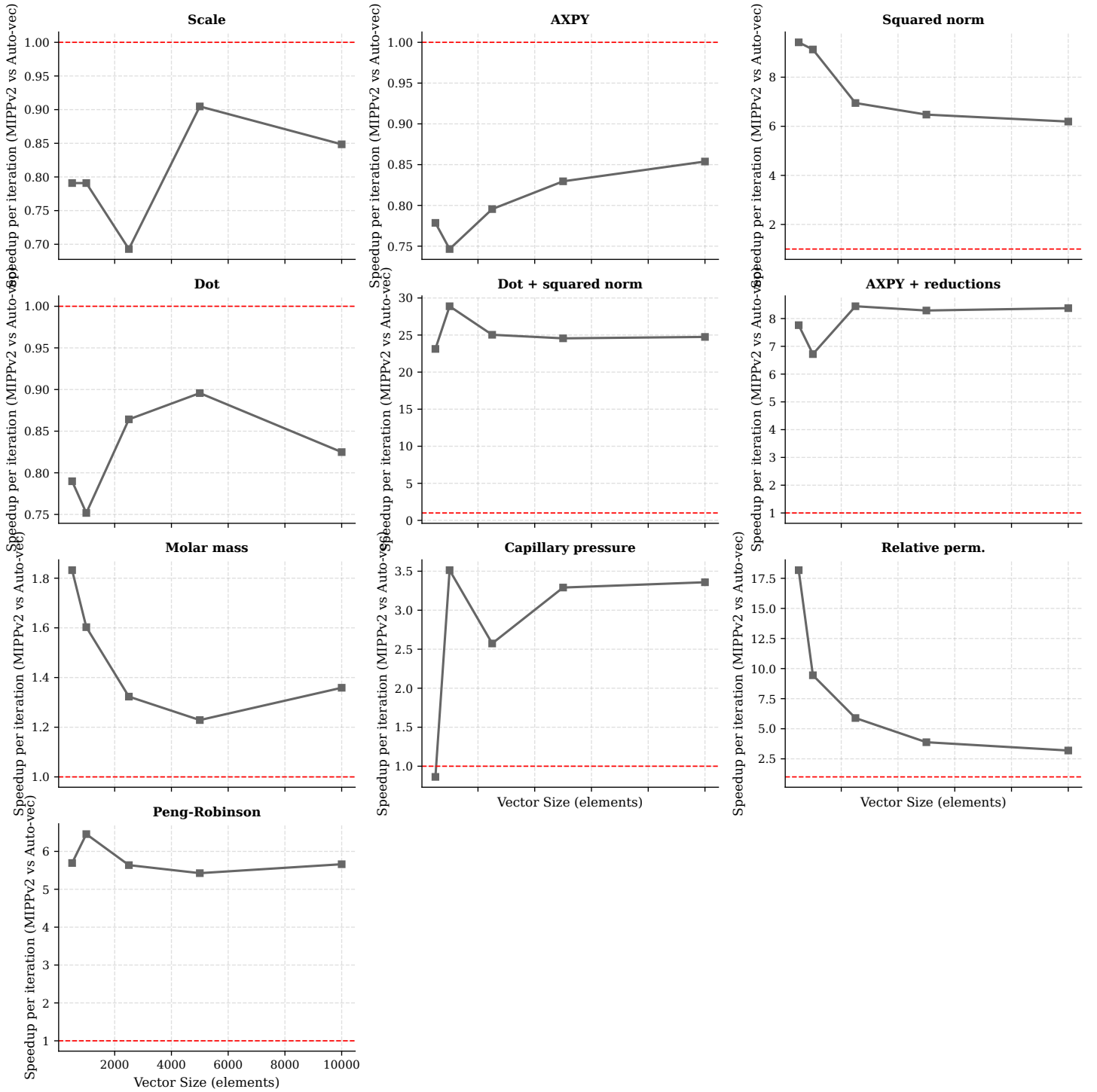


Fig. 20 – *Speedup* moyen par itération de chaque *kernel* en fonction de la taille des vecteurs d'entrée pour des flottants **double** précision sur Zen4.

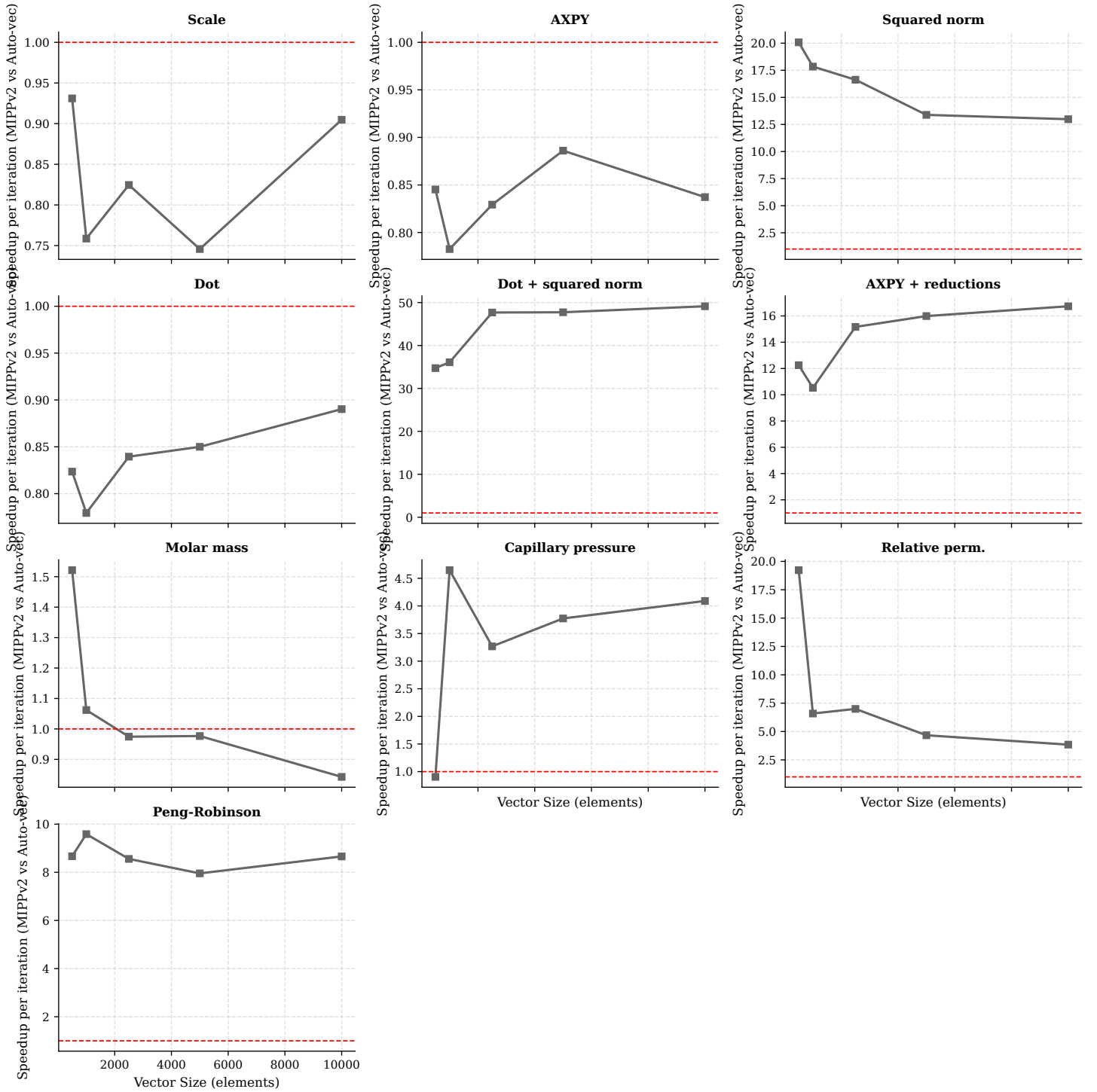


Fig. 21 – *Speedup* moyen par itération de chaque *kernel* en fonction de la taille des vecteurs d'entrée pour des flottants **simple** précision sur Zen4.

```

1  static void gemm_kernel(const Matrix<T>& A,
2                          const Matrix<T>& B,
3                          Matrix<T>& C)
4  {
5      // Découpage des matrices en blocs adaptés à la hiérarchie mémoire
6      for(size_t ib = 0; ib < A.rows; ib += BLOCK_M)
7      {
8          for(size_t jb = 0; jb < B.cols; jb += BLOCK_N)
9          {
10             // Bloc de sortie C de taille BLOCK_M × BLOCK_N
11             auto C_block = C.block(ib, jb);
12
13             for(size_t kb = 0; kb < A.cols; kb += BLOCK_K)
14             {
15                 // Blocs d'entrée :
16                 // A_block : BLOCK_M × BLOCK_K
17                 // B_block : BLOCK_K × BLOCK_N
18                 auto A_block = pack(A, ib, kb);
19                 auto B_block = pack(B, kb, jb);
20
21                 // Découpage du bloc en tuiles calculées
22                 // pour le micro-kernel
23                 for(size_t i = 0; i < BLOCK_M; i += MR)
24                 {
25                     for(size_t j = 0; j < BLOCK_N; j += NR)
26                     {
27                         ukernel(
28                             A_block.tile(i),
29                             B_block.tile(j),
30                             C_block.tile(i, j)
31                         );
32                     }
33                 }
34             }
35         }
36     }
37 }

```

Liste 21 – Illustration d'un *kernel* GEMM en C++.

```

1  auto c00 = set0<T>(); auto c01 = set0<T>();
2  auto c10 = set0<T>(); auto c11 = set0<T>();
3  auto c20 = set0<T>(); auto c21 = set0<T>();
4  auto c30 = set0<T>(); auto c31 = set0<T>();
5
6  const auto a0_ptr = A[i+0]; const auto a1_ptr = A[i+1];
7  const auto a2_ptr = A[i+2]; const auto a3_ptr = A[i+3];
8
9  for(size_t k = 0; k < A.cols; ++k)
10 {
11     const auto b0 = load<T>(B[k] + j);
12     const auto b1 = load<T>(B[k] + j + VL);
13
14     const auto a0 = set1<T>(a0_ptr[k]);
15     const auto a1 = set1<T>(a1_ptr[k]);
16     const auto a2 = set1<T>(a2_ptr[k]);
17     const auto a3 = set1<T>(a3_ptr[k]);
18
19
20     c00 = fmadd(a0,b0,c00); c01 = fmadd(a0,b1,c01);
21     c10 = fmadd(a1,b0,c10); c11 = fmadd(a1,b1,c11);
22     c20 = fmadd(a2,b0,c20); c21 = fmadd(a2,b1,c21);
23     c30 = fmadd(a3,b0,c30); c31 = fmadd(a3,b1,c31);
24 }

```

C++

Liste 22 – Extrait d'un des *kernels* optimisés pour Skylake.

```
1 struct ABlock4 {
2     double d0; double d1;
3     double d2; double d3;
4 };
```

C++

```
1 for(size_t i = 0; i < A.rows; i += MR) {
2     const auto* a0_ptr = reinterpret_cast<const ABlock4*>(A[i + 0]);
3     const auto* a1_ptr = reinterpret_cast<const ABlock4*>(A[i + 1]);
4     const auto* a2_ptr = reinterpret_cast<const ABlock4*>(A[i + 2]);
5
6     for(size_t j = 0; j < B.cols; j += NR) {
7
8         auto c00 = set0<T,lmul>(); auto c01 = set0<T,lmul>();
9         auto c10 = set0<T,lmul>(); auto c11 = set0<T,lmul>();
10
11         auto c20 = set0<T,lmul>(); auto c21 = set0<T,lmul>();
12
13         const size_t K4 = A.cols / 4; // On suppose A.cols % 4 == 0
14
15         for(size_t kk = 0; kk < K4; ++kk) {
16             const ABlock4 a0 = a0_ptr[kk];
17             const ABlock4 a1 = a1_ptr[kk];
18             const ABlock4 a2 = a2_ptr[kk];
19
20             auto b00 = load<T,lmul>(B[4*kk+0] + j);
21             auto b01 = load<T,lmul>(B[4*kk+0] + j + VL);
22
23             c00 = fmaddi(b00, a0.d0, c00); c01 = fmaddi(b01, a0.d0, c01);
24             c10 = fmaddi(b00, a1.d0, c10); c11 = fmaddi(b01, a1.d0, c11);
25             c20 = fmaddi(b00, a2.d0, c20); c21 = fmaddi(b01, a2.d0, c21);
26
27             // itération k+1
28             b00 = load<T,lmul>(B[4*kk+1] + j);
29             b01 = load<T,lmul>(B[4*kk+1] + j + VL);
30
31             c00 = fmaddi(b00, a0.d1, c00); c01 = fmaddi(b01, a0.d1, c01);
32             c10 = fmaddi(b00, a1.d1, c10); c11 = fmaddi(b01, a1.d1, c11);
33             c20 = fmaddi(b00, a2.d1, c20); c21 = fmaddi(b01, a2.d1, c21);
34
35             // etc
36         }
37
38         store(C[i + 0] + j, c00); store(C[i + 0] + j + VL, c01);
39         store(C[i + 1] + j, c10); store(C[i + 1] + j + VL, c11);
40         store(C[i + 2] + j, c20); store(C[i + 2] + j + VL, c21);
41     }
42 }
```

C++

Liste 23 – Définition de la struct ABlock4 et extrait d'un des kernels optimisés pour X100.

Les figures suivantes présentent le speedup des différentes implémentations de **micro-kernel** de GEMM sur Skylake et X100 par rapport aux versions Scalar IJK (*baseline* scalaire) et Auto-vectorized blocked (bonne version auto-vectorisée).

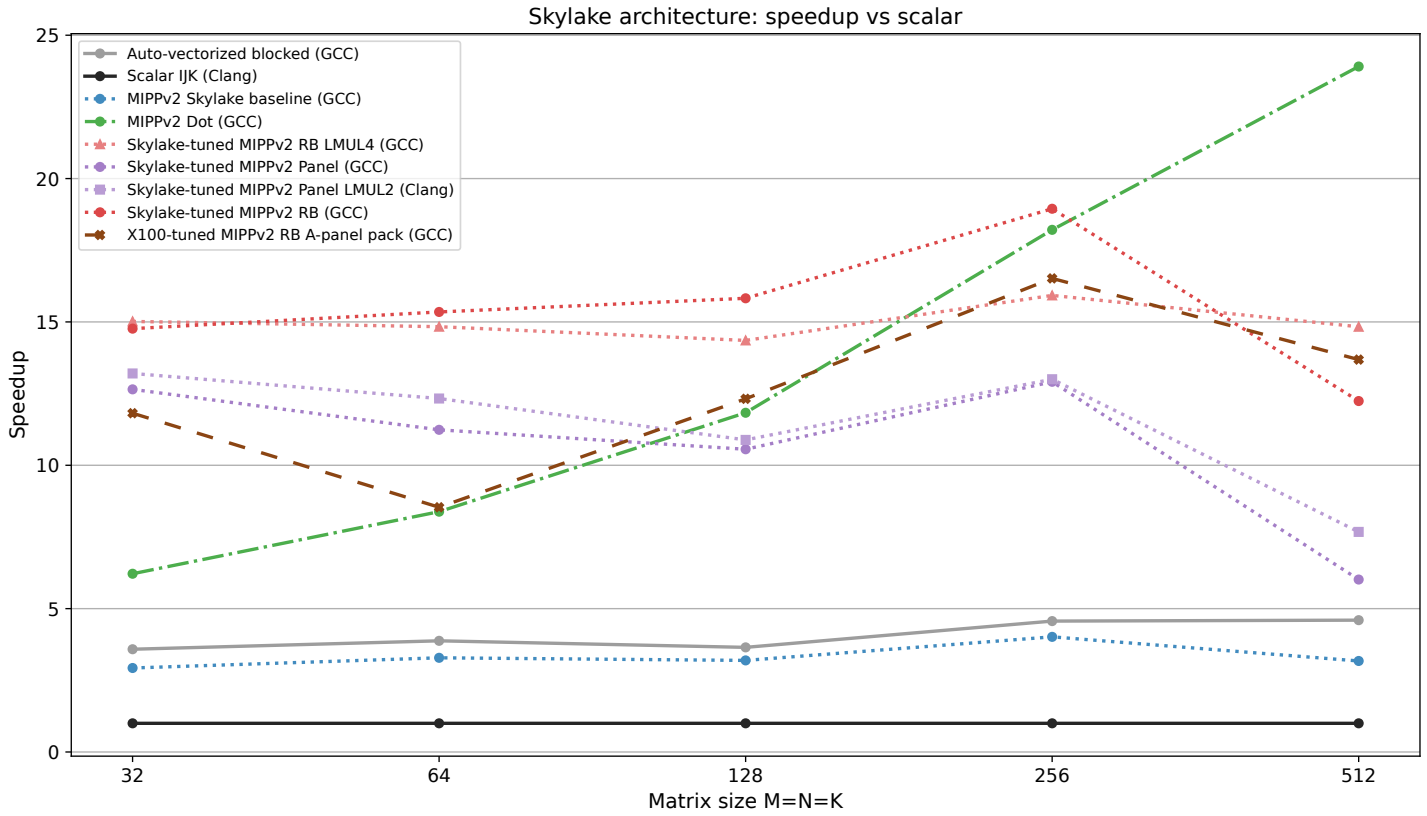


Fig. 22 – Speedup de différents **micro-kernels** de GEMM par rapport à la version Scalar IJK (Clang) sur Skylake.

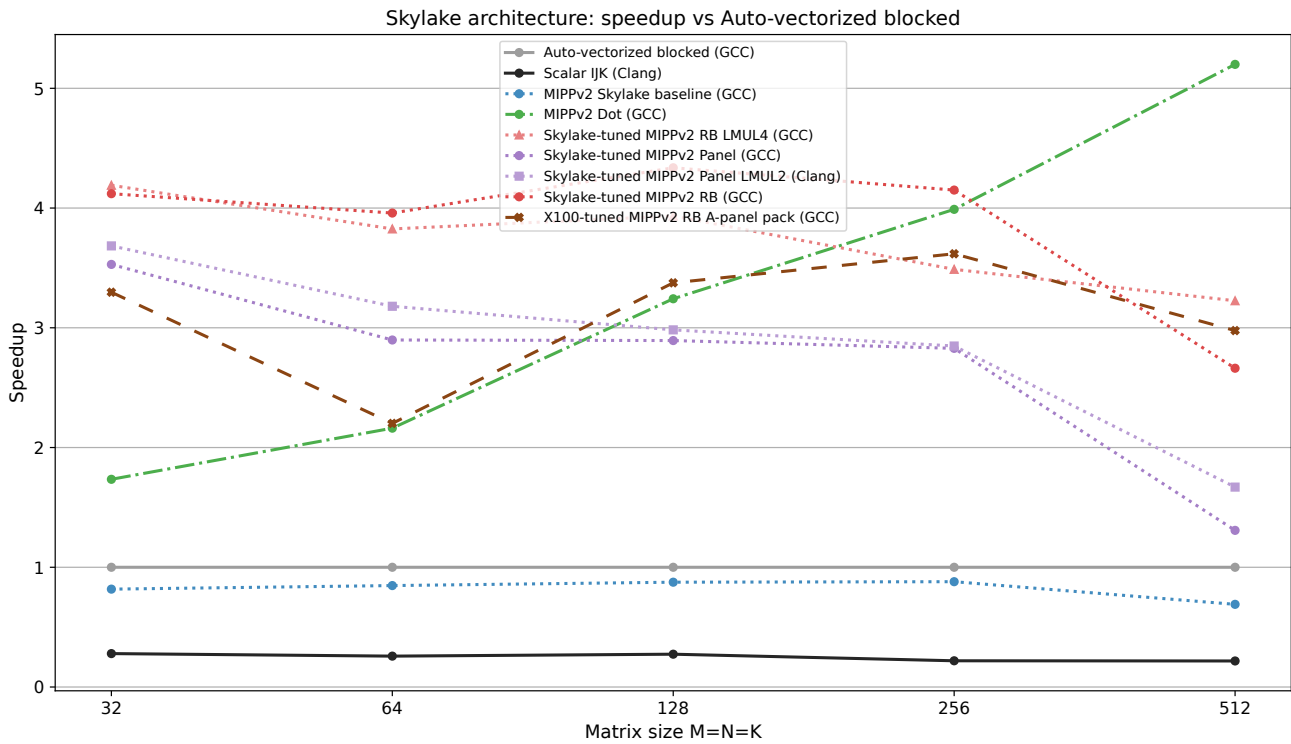


Fig. 23 – Speedup de différents **micro-kernels** de GEMM par rapport à la version Auto-vectorized blocked (GCC) sur Skylake.

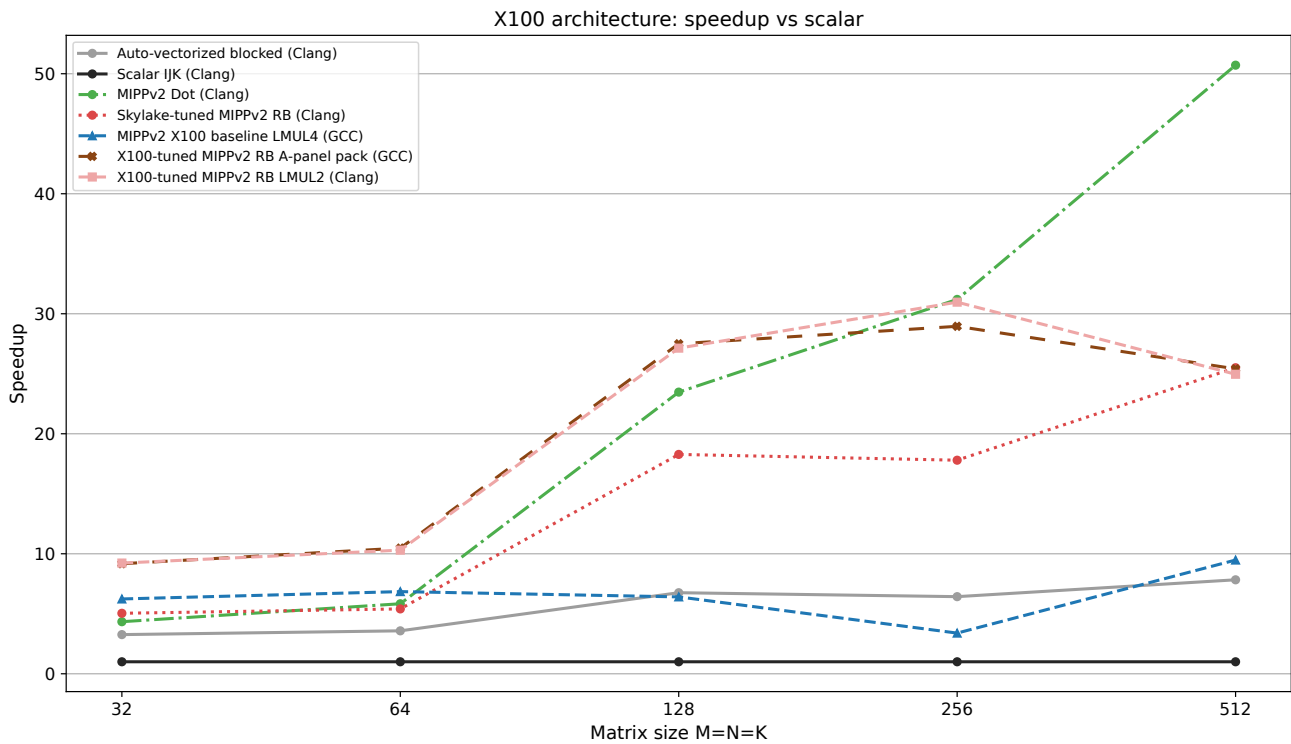


Fig. 24 – Speedup de différents **micro-kernels** de GEMM par rapport à la version Scalar IJK (Clang) sur X100.

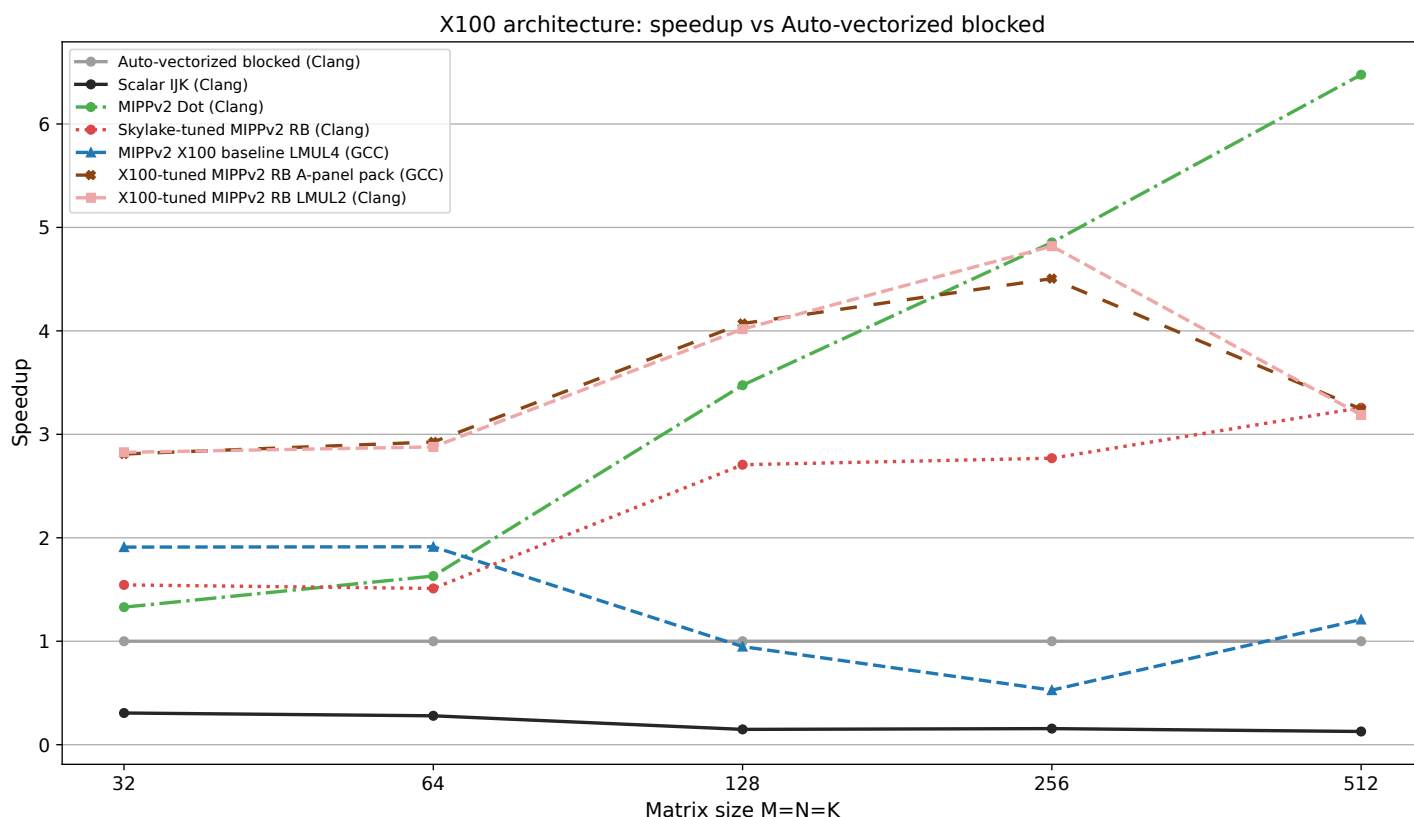


Fig. 25 – Speedup de différents **micro-kernels** de GEMM par rapport à la version Auto-vectorized blocked (Clang) sur X100.

Bibliographie

- [1] A. Cassagne, O. Aumage, D. Barthou, C. Leroux, et C. Jégo, « MIPP: a Portable C++ SIMD Wrapper and its use for Error Correction Coding in 5G Standard », in *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, in WPMVP'18. Vienna, Austria: Association for Computing Machinery, 2018. doi: [10.1145/3178433.3178435](https://doi.org/10.1145/3178433.3178435).
- [2] M. J. Flynn, « Some computer organizations and their effectiveness », *IEEE transactions on computers*, vol. 100, n° 9, p. 948-960, 1972.
- [3] « Guide de programmation SSE ». [En ligne]. Disponible sur: <https://www.intel.com/content/dam/develop/external/us/en/documents/d9156103-705230.pdf>
- [4] « Guide de programmation AVX512 ». [En ligne]. Disponible sur: <https://builders.intel.com/docs/networkbuilders/intel-avx-512-instruction-set-for-packet-processing-technology-guide-1645717553.pdf>
- [5] « Documentation arm NEON ». [En ligne]. Disponible sur: <https://developer.arm.com/documentation/102474/0100/Fundamentals-of-Armv8-Neon-technology>

- [6] « Whitepaper arm SVE ». [En ligne]. Disponible sur: <https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm-scalable-vector-extensions-and-application-to-machine-learning.pdf>
- [7] « Spécification de Risc-V "V" 1.0 ». [En ligne]. Disponible sur: https://docs.riscv.org/reference/isa/extensions/vector/_attachments/riscv-v-spec.pdf
- [8] « Documentation des instructions de configuration RVV ». [En ligne]. Disponible sur: <https://rvv-isadoc.readthedocs.io/en/latest/configure.html>
- [9] « Documentation des intrinsèques C de RVV ». [En ligne]. Disponible sur: <https://docs.riscv.org/reference/vector-c-intrinsics/rvv-intrinsic-spec.html>
- [10] « Documentation BPI-F3 ». [En ligne]. Disponible sur: https://docs.banana-pi.org/en/BPI-F3/BananaPi_BPI-F3
- [11] « Page web du Sipeed K3 Pico-ITX ». [En ligne]. Disponible sur: <https://sipeed.com/k3>
- [12] A. Cassagne, N. Amiot, et M. Bouyer, « Dalek: An Unconventional and Energy-Aware Heterogeneous Cluster », *arXiv preprint arXiv:2508.10481*, 2025.
- [13] « Documentation Spacemit K1 ». [En ligne]. Disponible sur: https://docs.banana-pi.org/en/BPI-F3/SpacemiT_K1
- [14] « Page d'informations générale de Spacemit ». [En ligne]. Disponible sur: <https://www.spacemit.com/about>
- [15] « Page web de la fiche technique du SpacemiT K1 sur le site de Banana Pi ». [En ligne]. Disponible sur: https://docs.banana-pi.org/en/BPI-F3/SpacemiT_K1_datasheet
- [16] « SpacemiT K3: A RVA23 RISC-V AI CPU with 60 TOPS AI Compute », technical report. [En ligne]. Disponible sur: <https://forum.spacemit.com/uploads/short-url/60aJ8cYNmrFWqHn4ddwwSzMLjlY.pdf>
- [17] « Page web de la fiche produit du SpacemiT K3 sur le site de SpacemiT ». [En ligne]. Disponible sur: https://www.spacemit.com/community/document/info?lang=en&nodepath=hardware/key_stone/k3/k3_docs/root_overview.md
- [18] « Page de docuementation de GNU Vector Extensions ». [En ligne]. Disponible sur: <https://gcc.gnu.org/onlinedocs/gcc-4.6.1/gcc/Vector-Extensions.html>
- [19] S. Brand et M. Kretz, « std::simd », technical report P1928R15, 2024. [En ligne]. Disponible sur: <https://wg21.link/P1928R15>
- [20] « Dépôt github d'Highway ». [En ligne]. Disponible sur: <https://github.com/google/highway>
- [21] « Dépôt github de VCL ». [En ligne]. Disponible sur: <https://github.com/vectorclass/version2>

-
- [22] « Site web de sleef ». [En ligne]. Disponible sur: <https://sleef.org/>
- [23] « Documentation Jinja2 ». [En ligne]. Disponible sur: <https://jinja.palletsprojects.com/en/stable/>
- [24] « Page GitHub de Catch2 ». [En ligne]. Disponible sur: <https://github.com/catchorg/Catch2>
- [25] J. Treibig, G. Hager, et G. Wellein, « Likwid: A lightweight performance-oriented tool suite for x86 multicore environments », in *2010 39th international conference on parallel processing workshops*, 2010, p. 207-216.
- [26] GEOSX Development Team, « GEOSX: An Open-Source Multi-Physics Reservoir Simulation Framework ». 2023. [En ligne]. Disponible sur: <https://geosx-geosx.readthedocs-hosted.com/>
- [27] M. L. Michelsen et J. M. Mollerup, *Thermodynamic Models: Fundamentals & Computational Aspects*, 2nd éd. Tie-Line Publications, 2007.
- [28] R. H. Brooks et A. T. Corey, « Hydraulic properties of porous media », *Hydrology Papers, Colorado State University*, vol. 3, p. 1-27, 1964.
- [29] A. T. Corey, « The interrelation between gas and oil relative permeabilities », *Producers Monthly*, vol. 19, n° 1, p. 38-41, 1954.
- [30] D.-Y. Peng et D. B. Robinson, « A new two-constant equation of state », *Industrial & Engineering Chemistry Fundamentals*, vol. 15, n° 1, p. 59-64, 1976.
- [31] « Site web de SLEEF ». [En ligne]. Disponible sur: <https://sleef.org/>
- [32] Y. Yang, « Dépôt github d'avx_mathfun ». [En ligne]. Disponible sur: https://github.com/reyoung/avx_mathfun