Stage de fin d'étude
Rapport final

Sujet : Support du jeu d'instructions vectoriel RISC-V dans la bibliothèque d'abstraction MIPP

Lou THIRION

Encadrants:

Roselyne CHOTIN et Adrien CASSAGNE

Table des matières

1. Remerciements 3
2. Contexte 3
   - 2.1. Les SIMD 3
   - 2.2. MIPP 5
   - 2.3. RISC-V 5
   - 2.4. RISC-V Vector extension 6
3. Objectif du stage 7
4. Analyse du problème et difficultés identifiées 7
   - 4.1. Comment fixer la taille des registres logiciels ? 7
   - 4.2. Implémentation des premières instructions 7
   - 4.3. Comment mesurer les performances ? 8
5. Étapes du projet 8
6. Procédure de recette 8
7. Le matériel utilisé : une Banana-pi 9
8. Implémentation 9
   - 8.1. Comment fixer la taille des registres 9
   - 8.2. Les fonctions intrinsèques 9
   - 8.3. Listes des instructions nécessaires 10
     - 8.3.1. Un cas très simple : la fonction `add_tab` 10
     - 8.3.2. L'algorithme de Mandelbrot 10
9. Comment mesurer les performances ? 12
10. Résultats 12
    - 10.1. add_tab 12
      - 10.1.1. Validation fonctionnelle 12
      - 10.1.2. Version non vectorisée et Version auto-vectorisée 13
      - 10.1.3. Version intrinsics RVV et Version MIPP 14
      - 10.1.4. Le paramètre LMUL 16
    - 10.2. Mandelbrot 18
      - 10.2.1. Validation fonctionnelle 18
      - 10.2.2. Premières versions vectorielles 18
      - 10.2.3. Deuxième version 21
      - 10.2.4. Mandelbrot avec LMUL > 1 23
11. Récapitulatif de la procédure de recette 27
12. Conclusion 28
13. Bibliographie 28

1. Remerciements

Mon stage s'est déroulé au LIP6, le laboratoire informatique de Sorbonne Université, au sein de l'équipe ALSOC, j'aimerais les remercier de m'avoir accueilli pour mon stage de fin d'étude. Parmi eux, j'aimerais remercier particulièrement mes encadrants Adrien et Roselyne pour avoir accepté de me faire confiance, pour leur bienveillance et pour avoir su me remettre sur les rails quand je commençais à me disperser. Je remercie également mes camarades de bureau, Yacine, Ali, Noé, Maxime et Rémi pour leurs précieux conseilles et pour toutes les discussions passionnantes qu'ont a pu avoir. Merci à mes amis qui étaient en stage aussi au lip6 qui m'ont aider à me changer les idées un grand nombre de fois le temps de la pose du midi. Enfin merci à ma copine Mélo pour être toujours là, en particulier quand la motivation est au plus bas.

2. Contexte

2.1. Les SIMD

Depuis leurs débuts dans les années 70, les performances des microprocesseurs ont augmenté de manière exponentielle, principalement grâce à deux facteurs qui sont l'augmentation de la fréquence d'horloge des processeurs et l'augmentation du nombre de transistors sur une même surface de silicium.

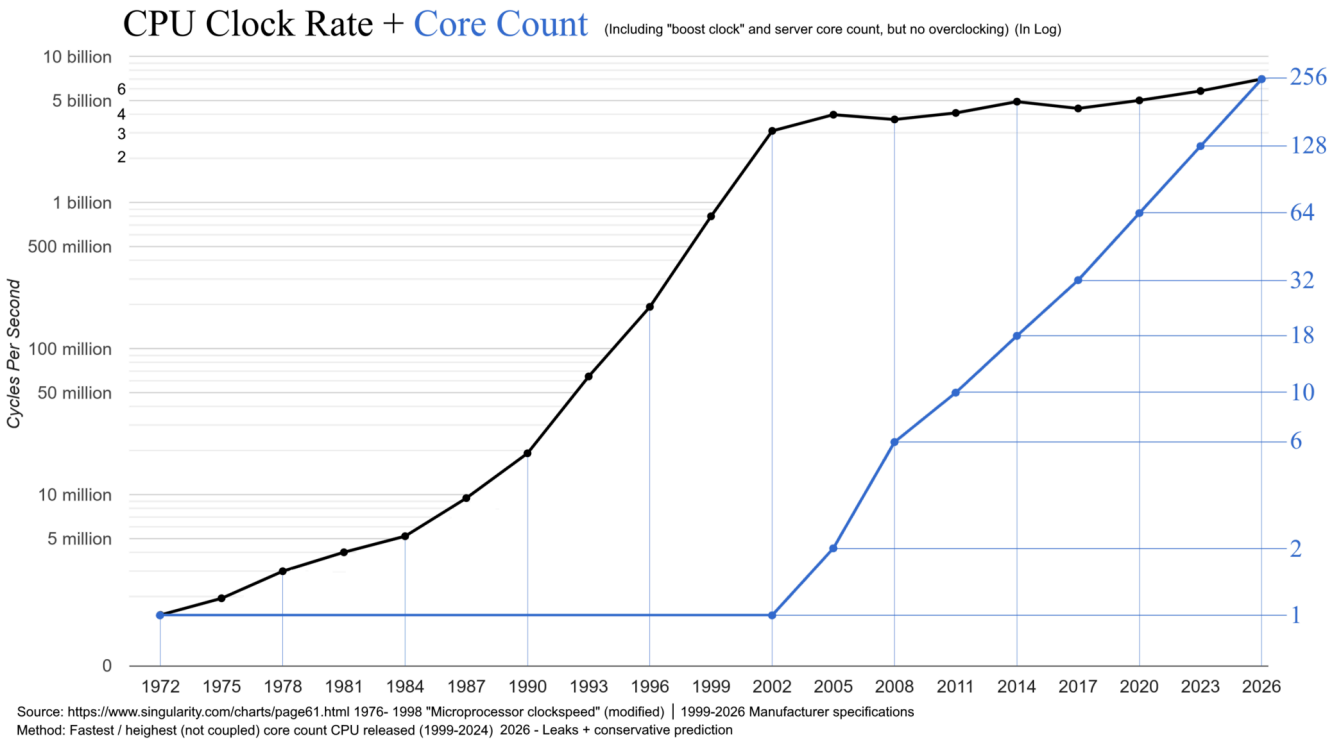

Fig. 1. – Évolution de la fréquence d'horloge des processeurs depuis 1972

Sur la Fig. 1, on remarque que la fréquence des processeurs cesse d'augmenter de manière significative à partir du début des années 2000. Ce plateau technologique soulève la question suivante : comment améliore-t-on les performances des processeurs sans augmentation de leur vitesse ?

La réponse se trouve dans **la loi de Moore**.

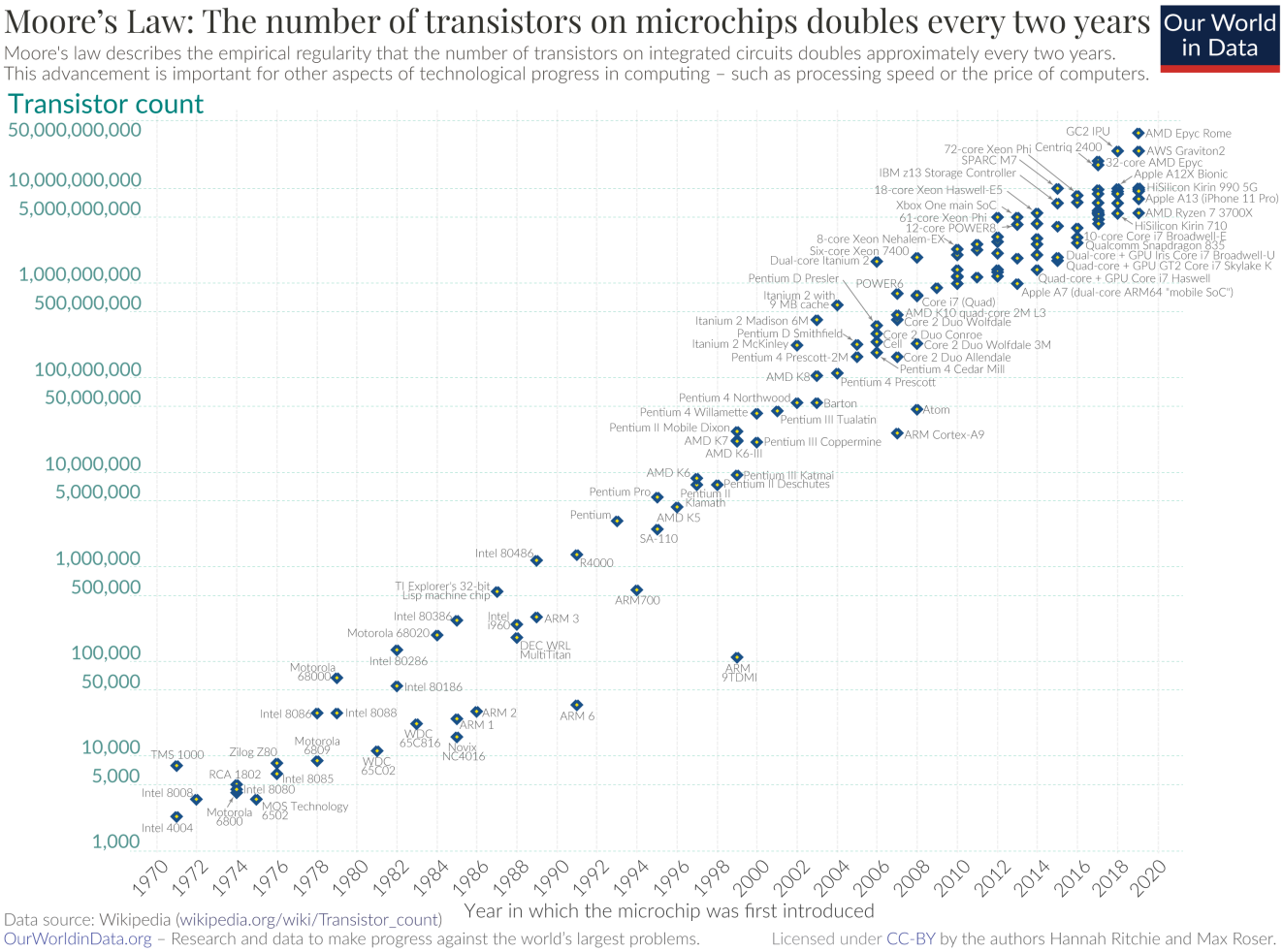

Fig. 2. – Loi de Moore

En effet, la loi de Moore (en hommage à Gordon Moore) prédit un doublement du nombre de transistors sur une même surface de silicium. Comme on peut le voir sur la Fig. 2, cette loi est restée vraie jusqu'au milieu des années 2010 où les fabricants de puces électroniques arrivent proche des limites physiques des matériaux utilisés.

Grâce à ce matériel disponible en quantité toujours plus grande, la solution pour continuer d'augmenter les performances des processeurs est donc de paralléliser les exécutions. Ces tâches vont alors exploiter plus de matériel simultanément, ce qui permet d'améliorer le débit d'exécutions des machines.

Il existe 3 niveaux de parallélismes :

- **niveau instructions**, c'est ce niveau qu'exploitent les processeurs superscalaires modernes.
- **niveau données**, avec les SIMD (Single Instruction Multiple Data) que nous allons voir.
- **niveau tâches**, avec les processeurs multi-cœurs.

Nous nous intéresserons ici au parallélisme niveau données car mon stage se porte sur **les instructions SIMD** (aussi appelées **instructions vectorielles**). La Fig. 3 montre bien le principe des SIMD, une instruction (Single Instruction) est appliquée à un ensemble de données (Multiple Data). Le nombre de cycle d'horloge par instruction vectorielle est généralement inférieure au nombre de données contenus dans le vecteur, ce qui permet aux instructions vectorielles d'être plus performantes que les instructions scalaires (moins de cycles d'horloge par données calculées autrement dit meilleur de débit de données).

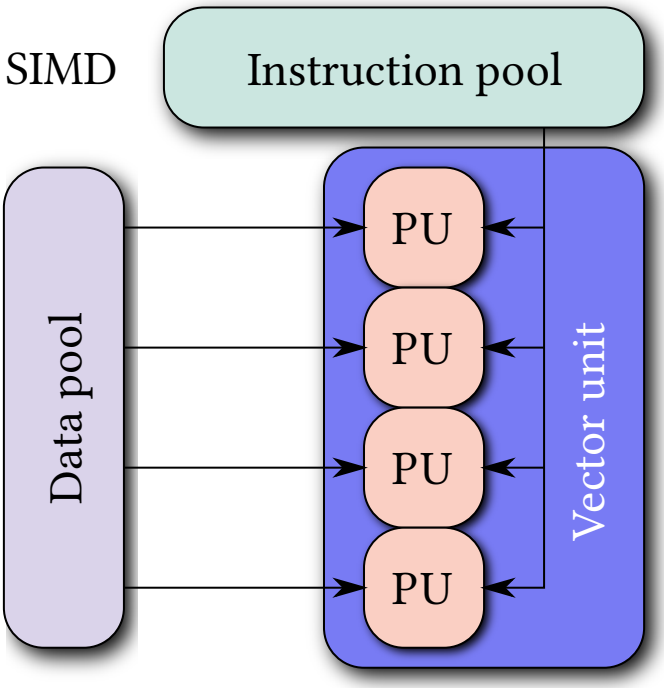

Fig. 3. – Single Instruction Multiple Data

Les instructions SIMD sont généralement regroupées dans des extensions matérielles et leur utilisation peut permettre des gains de performance importants mais elles nécessitent des adaptations logicielles (utiliser les instructions vectorielles à la place des instructions scalaires). Ces adaptations peuvent être faites de deux manières.

La première est la vectorisation automatique par le compilateur. Malheureusement ,et comme nous le verront plus loin, on constate que les compilateurs d'aujourd'hui ne sont capables de vectoriser que des calculs très simples.

La seconde manière de faire est que le développeur logiciel utilise lui-même les instructions vectorielles via des macros fournies par les fabriquants de processeurs et reconnues par les compilateurs. Ces macros sont appelées « fonctions intrinsèques » et chacune d'elle correspond le plus souvent à une instruction assembleur. Malheureusement, là encore il y a des difficultés car chaque fabriquant a ses propres instructions avec ses propres fonctions intrinsèques, et pour chaque extension vectorielle il en existe un très grand nombre, ce qui rend très compliqué l'écriture de code lisible et portable.

2.2. MIPP

Un compromis existe entre la simplicité d'utilisation des compilateurs et les performances obtenues avec les fonctions intrinsèques grâce aux **wrappers logiciels**, qui offrent une interface simple qui s'abstrait de l'architecture matérielle mais qui permet de vectoriser à la main de manière efficace avec un surcoût relativement faible. Il existe de nombreux wrappers logiciels pour les SIMD, dans le cadre de ce stage, je travaille sur le wrapper **MIPP** (My Intrinsics Plus Plus) qui est un projet open-source, basé sur les templates du C++.

MIPP est implémenté pour la plupart des extensions SIMD qui sont utilisées dans les architectures modernes mais pas encore pour l'extension vectorielle du RISC-V.

2.3. RISC-V

RISC-V est une architecture matérielle généraliste dont le jeu d'instructions est ouvert. L'*Instruction Set Architecture* (ISA) comporte une base généraliste (les instructions minimums pour qu'une

machine soit fonctionnelle) sur laquelle se greffent des extensions comme par exemple la FPU (l'unité de calcul sur les flotants) ou encore l'extension vectorielle qui va nous intéresser ici.

2.4. RISC-V Vector extension

D'un point de vu matérielle, toutes les extensions vectorielles comportent un banc de registres spécifique, un vecteur correspond à un registre de ce banc. La plupart des extensions vectorielles ont une taille de registre fixe et qui ne peut pas être modifié au moment de l'implémentation matérielle. Comme on peut le voir sur la Fig. 4 avec les extensions de chez Intel, où par exemple l'extension AVX, quelque soit sont implémentation, a des registres de 256 bits. Au contraire, l'extension V de RISC-V (abrégé en RVV) a la particularité d'être « vector length agnostic ». Autrement dit, le nombre d'élément par vecteur n'est pas définie par l'ISA, mais doit être définie lors de l'implémentation matérielle, ce nombre n'est donc connue par le logiciel qu'au moment de l'exécution.

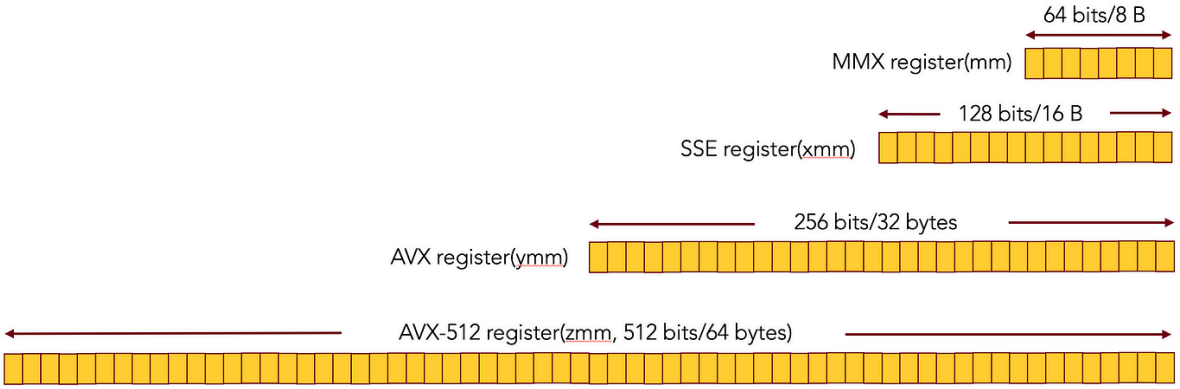

Fig. 4. – Les tailles de registres vectoriels pour différentes extensions d'Intel

Fig. 5. – Vecteur dont le nombre d'élément n'est connu qu'à l'exécution avec VLEN

Cette particularité renforce la généricité de RISC-V car RVV est aussi bien adaptée pour des machines de bureau avec des tailles de vecteurs infèrieures à 1024 bits qu'à des super-calculateurs avec des vecteurs de plusieurs Kilobits.

Une autre particularité de RVV est la possiblité de combiner plusieurs registres vectoriels pour en former un plus gros. Cette propriété est ajustable avec le paramètre LMUL qui définit le nombre de registres à combiner comme on peut le voir sur la Fig. 6.

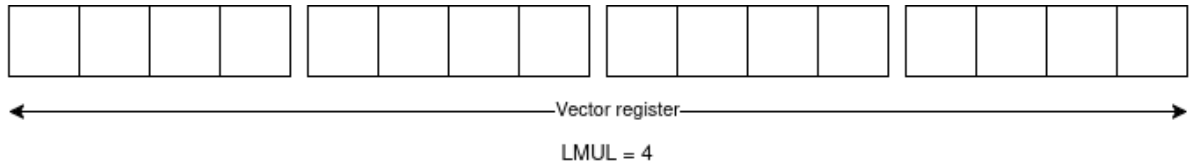

Fig. 6. – Avec LMUL=4, 4 registres vectoriels sont combinés en un seul

LMUL peut également être inférieur à 1 pour permettre de n'utiliser qu'une partie d'un registre vectoriel (Fig. 7).

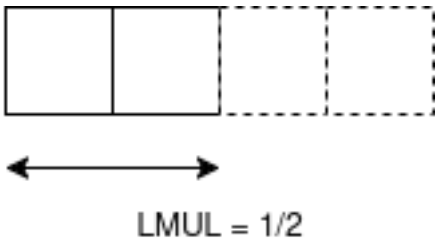

Fig. 7. – Avec LMUL=1/2, seulement la moitié du registres vectoriel est utilisée

La fonctionnalité de regrouper ou diviser les registres vectoriels peut être utile si, dans une même boucle, on manipule plusieurs tableaux avec des éléments de tailles différentes. Elle peut également servir pour faire du déroulage de boucle.

3. Objectif du stage

Nous avons vu plus haut que les SIMD permettent des gains de performance mais nécessitent des adaptations logicielles qui peuvent être faites à l'aide de wrappers logiciels comme MIPP. Par ailleurs RISC-V se positionne comme une alternative open-source sérieuse aux géants de l'Hardware comme Intel ou ARM. **L'objectif de ce stage est donc de proposer une première implémentation du wrapper MIPP pour RVV, l'extension vectorielle du RISC-V.**

Cette première implémentation ne sera pas complète mais aura plutôt comme but d'essayer de montrer les avantages et les limites de MIPP sur cette nouvelle architecture. Elle permettra également de poser quelques bases sur lesquelles s'appuyer pour les futures évolutions du wrapper (prise en charge complète de RVV, génération automatique du wrapper).

4. Analyse du problème et difficultés identifiées

4.1. Comment fixer la taille des registres logiciels ?

Une première difficulté à surmonter dans ce projet est que MIPP n'a pas été conçu pour des registres « length agnostic ». Pour ajouter à MIPP une nouvelle extension vectorielle il est nécessaire de spécifier explicitement dans le code la taille des registres vectoriels. Or comme nous l'avons vu, cette taille est propre à une implémentation matérielle et ne peut donc pas être connue à la compilation.

4.2. Implémentation des premières instructions

Une fois la taille des registres logiciels fixée, nous pourrons commencer à implémenter les premières instructions MIPP pour l'extension vectorielle du RISC-V. Ce qui nous permettra ensuite de mesurer les performances de MIPP sur cette architecture.

4.3. Comment mesurer les performances ?

Étant donné que la finalité est de comparer les performances du wrapper MIPP avec d'autres implémentations, nous devons au préalable déterminer les mesures à effectuer et les méthodes de comparaison que nous allons utiliser. Il faut également choisir comment effectuer la mesure correctement et donc établir un protocole à appliquer de manière identique pour chaque implémentation.

5. Étapes du projet

Pour résoudre les différents problèmes que nous venons d'énoncer, une première étape sera donc de définir ce qui doit être mesuré et comparé, et d'implémenter un banc de test qui applique un protocole permettant de limiter au maximum les incertitudes et de donner les performances d'un programme.

Une seconde étape consistera à fixer dans le code de MIPP la taille des registres logiciels en spécifiant directement au compilateur la taille des vecteurs dans le matériel utilisé.

Enfin la dernière étape sera d'ajouter les premières instructions RVV dans MIPP. Pour faire cela, une première partie sera d'ajouter les instuctions minimales (load, add, store) pour une application extrêmement simpliste (addition vectorielle). Ce premier palier permettra d'avoir une base fonctionnelle solide pour la suite et de commencer à comparer les performances de MIPP par rapport à d'autres implémentations (scalaire, auto-vectorisée, vectorisée avec les fonctions intrinsèques).

La seconde partie consistera à ajouter de nouvelles instructions RVV dans MIPP pour une application plus complexe, l'algorithme de Mandelbrot. Les instructions nécessaires seront plus variées et plus nombreuses que pour la première partie.

Pour ajouter de nouvelles instructions RVV dans MIPP, la procédure à suivre sera la suivante:
- Lister l'ensemble des instructions vectorielles nécessaires pour l'application
- Et pour chacune d'elles :
  - l'ajouter à MIPP
  - la tester
- Évaluer les performances de l'application avec MIPP
- Et enfin comparer les performances avec d'autres implémentations

6. Procédure de recette

L'objectif de ce stage est de proposer une première implémentation du wrapper MIPP pour RVV. Afin de valider la réussite ou non de cet objectif, les applications qui auront été implémentées avec MIPP devront, dans un premier temps, passer un **test fonctionnel** pour être sûr que les résultats obtenus entre les différentes implémentations (scalaires, auto-vectorisé, ...) soient comparables. Ce test consistera à comparer la sortie de la version MIPP avec la sortie d'une version scalaire de référence et les deux devront être strictement identiques.

Dans un second temps, il sera intéressant d'**observer les performances des implémentation MIPP des différentes applications** (addition vectorielle, algorithme de Mandelbrot) comparées aux versions vectorisées par le compilateur (si celui-ci réussi à vectoriser automatiquement) et une version vectorisée à la main avec les fonctions intrinsèques. Le résultat souhaitable et attendu est que la version MIPP ait des performances meilleurs que la version auto-vectorisée par le compilateur (encore une fois, si le compilateur réussi à vectoriser automatiquement). Les performances de la version MIPP devront également être du même ordre de grandeur que les performances de la version

vectorisée à la main avec les fonctions intrinsèques. Si c'est le cas, alors nous aurons montré que, pour cette application, MIPP permet de gagner en performance avec les SIMD sans perdre en lisibilité et en portabilité.

Une autre contribution intéressante serait d'évaluer également les performances des différentes implémentations avec des valeurs de LMUL supérieures à 1. On pourrait s'attendre à ce que les performances soient améliorées significativement (amélioration d'au moins 10% avec des versions avec LMUL>1 par rapport à des versions avec LMUL=1).

7. Le matériel utilisé : une Banana-pi

Afin d'obtenir les performances réelles de RVV et MIPP sur du matériel, j'avais à ma disposition pendant ce stage une Banana-pi BPI-F3 comportant une implémentation du RISC-V avec une extension RVV 1.0. Cette extension vectorielle comporte des registres de 256 bits, étant donné que j'ai principalement utilisé des éléments de 32 bits (`int32` et `float32`), les registres me permettent d'avoir des vecteurs de 8 éléments. Une remarque importante à faire est que la Banana-pi est l'une des seuls implémentations matérielles du RISC-V contenant une extension RVV 1.0 que l'on peut trouver facilement sur le marché.

Tous les tests ayant été faits sur cette carte, l'ensemble des performances mesurées valent uniquement pour cette architecture. Cela permet néanmoins de montrer l'intérêt de RVV et de MIPP par rapport à des implémentations scalaires ou vectorisées par le compilateur.

8. Implémentation

8.1. Comment fixer la taille des registres

Une des difficultés identifiées plus haut est de fixer la taille des registres de RVV pour qu'ils soient adaptés à MIPP. Une première manière de faire a été de spécifier de manière statique la taille des registres. Comme vu plus haut, la Banana-pi contient une extension RVV avec des registres de 256 bits, on peut donc spécifier dans MIPP que les registres logiciels font 256 bits.

Cette manière de faire a ses limites, si on veut faire fonctionner le wrapper sur une implémentation de RVV différente alors il faut adapter cette valeur. Pour que ce soit plus dynamique on utilise la macro `__riscv_v_fixed_vlen` dont la valeur dépend du paramètre `-march` spécifié à la compilation.

Par exemple, pour la Banana-pi, la valeur du paramètre `-march` à spécifier est la suivante :

```
1  -march=rv64gv1_zve32f_zvl256b
```

8.2. Les fonctions intrinsèques

Avant de parler des différentes instructions vectorielles que j'ai utilisées, il est nécessaire de présenter les fonctions intrinsèques et leur nomenclature.

Les fonctions intrinsèques servent d'interface bas niveau pour les instructions assembleur de RVV. Elles permettent une utilisation efficace de RVV avec du code relativement clair et explicite au besoin (mais pas forcément facile à lire).

Les noms des fonctions sont structurés de manière assez simple, ils commencent toujours par `__riscv` suivi du nom de l'opération (`_vadd`, `_vmul`, ...). Ensuite soit la fonction est implicite et c'est au compilateur de déduire les types des paramètres d'entrée et de sortie, soit la fonction est explicite : son nom est alors suivi des informations concernant les types d'entrée et de sortie.

Opération	type	LMUL	vtype	Implicite	Explicite
add	unsigned int32	1	`vuint32m1_t`	`__riscv_vadd`	`__riscv_vadd_vv_u32m1`
mul	int64	2	`vint64m2_t`	`__riscv_vmul`	`__riscv_vmul_vv_i64m2`
load	float16	1/2	`vfloat16mf2_t`	`__riscv_vle16`	`__riscv_vle16_v_f16mf2`
set	int32	8	`vint32m8_t`	`__riscv_vmv_v`	`__riscv_vmv_v_v_i32m8`

Tableau 1. – Quelques exemples de fonctions intrinsèques de RVV

Comme on peut le voir dans le Tableau 1, la forme explicite d'une fonction intrinsèque comporte une ou deux lettres (ici `_v` ou `_vv`) pour indiquer si les paramètres de la fonction sont des vecteurs ou des scalaires (lettre `x` pour un scalaire de type entier et `f` pour un scalaire de type `float`). Elle comporte également le type de vecteur manipulé, par exemple `_u32m1` signifie que l'opération s'effectue sur des entiers non signés de 32 bits.

8.3. Listes des instructions nécessaires

Avant d'implémenter les premières instructions MIPP pour RVV, j'ai commencé par lister les instructions dont j'avais besoin en fonction de l'application que je voulais tester.

8.3.1. Un cas très simple : la fonction `add_tab`

La première application, qui nous permet d'effectuer les premiers tests de fonctionnement et de performance de RVV et de MIPP, est la fonction `add_tab`. Comme on peut le voir Liste 1, cette fonction fait une simple addition entre les éléments de deux tableaux et stocke le résultat dans un troisième.

```c
1 void add_tab(int n, int32_t *a, int32_t *b, int32_t *res){
2   for(int i = 0; i<n; i++)
3       res[i] = a[i] + b[i];
4 }
```

Liste 1. – Code C de la fonction `add_tab`

Les calculs se font sur des entiers de 32 bits et ne nécessitent que 3 instructions vectorielles pour fonctionner :

type d'instruction	instruction MIPP	insruction RVV
lecture mémoire	`load`	__riscv_vle32
écriture mémoire	`store`	__riscv_vse32
addition	`add` ou +	__riscv_vadd

8.3.2. L'algorithme de Mandelbrot

La deuxième application, plus complexe, qui va nous permettre de tester les performances des instructions vectorielles est l'algorithme de Mandelbrot.

Comme on peut le voir dans le code Liste 2, la fonction `mandelbrot` est plus calculatoire, elle a besoin d'instructions plus variées. Les calculs se font principalement sur des `float` de 32 bits.

On remarque que la boucle `while` (ligne 9 du code Liste 2) impose une condition pour continuer ou non les calculs. Cette condition d'arrêt varie en fonction des valeurs de `x` et de `y`, elle est donc différente pour chaque élément du vecteur. Pour faire en sorte qu'une opération soit effectuée seulement sur une partie du vecteur, on utilise des opérations masquées. Comme on peut le voir sur la Fig. 8, ces opérations permettent de sélectionner les éléments du vecteur destination qui reçoivent le résultat de l'opération, les autres éléments ne sont pas pris en compte.

Au final la fonction `mandelbrot` va nécessiter les instructions vectorielles suivantes :

type d'instruction	instruction MIPP	insruction RVV
initialisation avec constante	=	__riscv_vfmv
addition	`add` ou +	__riscv_vfadd
soustraction	`sub` ou -	__riscv_vfsub
multiplication	`mul` ou *	__riscv_vfmul
fused multiplication and addition	`fmadd`	__riscv_fmacc
conversion float vers int	`cvt`	__riscv_vfcvt_x
comparaison	`cmplt`	__riscv_vmflt
réduction sur masque	`testz`	__riscv_vfirst
écriture mémoire	`store`	__riscv_vse32

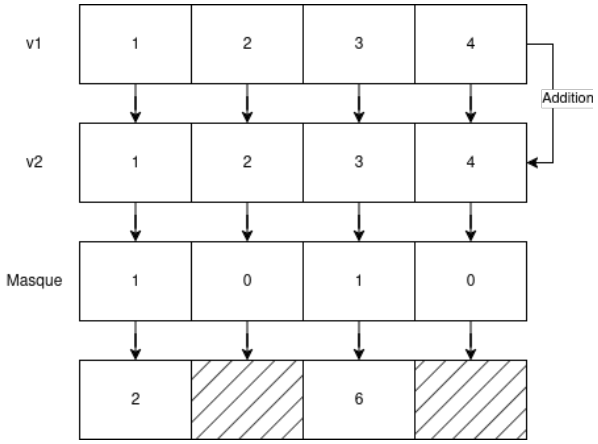

Fig. 8. – Schéma d'une opération masquée

```c
void mandelbrot(size_t width, size_t maxIter, uint32_t *res) {
  for (size_t y = 0; y < width; ++y)
  for (size_t x = 0; x < width; ++x) {
    float cx = x * 2.0f / width - 1.5;
    float cy = y * 2.0f / width - 1;
    size_t iter = 0;
    float zx = 0, zy = 0, zxS = 0, zyS = 0;

    while (zxS + zyS <= 4 && iter < maxIter) {
      zxS = zxS - zyS + cx;
      zy = 2 * zx * zy + cy;
      zx = zxS;
      zxS = zx*zx;
      zyS = zy*zy;
      ++iter;
    }
    *res++ = iter;
  }
}
```

Liste 2. – Code C de la fonction `mandelbrot`

9. Comment mesurer les performances ?

Pour pouvoir comparer les performances des différentes versions implémentées, il est nécessaire de décider au préalable de quelles sont les mesures à effectuer. Ce sont ces mesures qui doivent nous donner une idée la plus précise possible des performances de chaque implémentation afin de les comparer et de valider ou non l'efficacité de RVV et de MIPP.

Comme nous le verrons par la suite, la fonction `add_tab` accède un grand nombre de fois à la mémoire et ce sont surtout ces accès qui vont limiter les performances. Nous nous intéresserons donc au **débit de données** qui pourront être produites par la fonction (nombre d'octet additionné par seconde) en fonction de la taille des tableaux d'entrée.

Comme nous l'avons vu plus haut, la fonction `mandelbrot` est plus calculatoire. Les accès mémoire sont moindres par rapport à la première application. Les débits de données mesurés en fonction de la taille des données calculées devrait donc être constant. Il sera plus pertinent ici de s'intéresser au **speedup**, c'est à dire le pourcentage de gain entre le temps d'exécution d'une implémentation de référence et le temps d'exécution des autres implémentations.

10. Résultats

10.1. add_tab

10.1.1. Validation fonctionnelle

Pour chaque nouvelle implémentation de la fonction `add_tab`, je me suis assuré qu'elle fonctionne correctement et de manière identique à la version scalaire qui me sert de référence. Pour cela, j'ai comparé les valeurs du tableaux de sortie, ces valeurs devaient être strictement identiques aux valeurs de la version de référence.

10.1.2. Version non vectorisée et Version auto-vectorisée

Dans un premier temps, on regarde quelles sont les performances de la vectorisation automatique du compilateur par rapport à une version non vectorisée.

Dans la colonne de gauche du Tableau 4, on peut voir le code de la version non vectorisée avec, lignes 6 et 7, les instructions `load` permettant d'accéder aux éléments du tableau, l'addition ligne 10 et l'instruction `store` ligne 11 pour écrire dans le tableau de sortie.

De la même manière, dans la colonne de droite on a les `vload` de 32 bits (`vle32.v`) ligne 5 et 6, l'addition vectorielle (`vadd.vv`) ligne 11 et le `vstore` (`vse32.v`) ligne 12.

Ces deux codes se ressemblent beaucoup, on remarque qu'ils font les mêmes opérations (lecture mémoire, addition, écriture mémoire), on peut néanmoins noter l'instruction vsetvli, à la ligne 4 de la colonne de droite, qui permet de fixer la taille du vecteur en fonction de plusieurs paramètres, notamment le nombre d'éléments qu'il reste à calculer, la taille des éléments ou encore le paramètre lmul.

```asm
1  add_tab:
2    ble a0,zero,.L7
3    slli  a0,a0,2
4    add a0,a1,a0
5  .L5:
6    lw  a4,0(a1)
7    lw  a5,0(a2)
8    addi  a1,a1,4
9    addi  a2,a2,4
10   addw  a5,a5,a4
11   sw  a5,0(a3)
12   addi  a3,a3,4
13   bne a1,a0,.L5
14 .L7:
15   [...]
```

```asm
1  add_tab_autovec:
2    ble a0,zero,.L12
3  .L10:
4    vsetvli a5,a0,e32,m1,ta,ma
5    vle32.v v2,0(a1)
6    vle32.v v1,0(a2)
7    slli  a4,a5,2
8    sub a0,a0,a5
9    add a1,a1,a4
10   add a2,a2,a4
11   vadd.vv v1,v1,v2
12   vse32.v v1,0(a3)
13   add a3,a3,a4
14   bne a0,zero,.L10
15 .L12:
16   [...]
```

Tableau 4. – Code assembleur RISC-V des versions non vectorisée et auto-vectorisée de la fonction add_tab

On observe ensuite dans la Fig. 9, la différence importante de performance entre les deux versions. La courbe bleue représente les performances de la version non vectorisée et on peut voir qu'elles sont indépendantes de la taille des tableaux avec des valeurs légèrement au dessus de 1 Go/s.

Parallèlement, la courbe jaune, représentant les performances de la version vectorisée, oscille beaucoup plus en fonction de la taille des tableaux. Pour des tableaux de 0 à 16 Koctet, on remarque que les performances sont d'environ 4.3 Go/s, soit 4 fois les performances de la version non vectorisée. Pour des tableaux plus grands la version vectorisée oscille de 0.5 à 2.5 Go/s et reste globalement meilleure que la version non vectorisée. La différence de performance entre les deux versions s'explique grâce aux instructions vectorielles qui permettent de faire des calculs sur un vecteur de données en parallèle, d'économiser des itérations et donc d'augmenter le débit.

Pour la version vectorisée, on peut se demander d'où vient la différence de performance avec des tableaux inférieurs à 16 Ko et avec des tableaux plus grands. Cela s'explique avec les accès mémoire et le cache car, dans `add_tab`, les instructions d'accès à la mémoire représentent une partie

importante du code de la fonction et peuvent donc constituer un facteur limitant important. Étant donné que la fonction est exécutée un grand nombre de fois, lors de la première exécution, lorsque les tableaux sont inférieurs à 16 Ko, une grande partie des données (voir la totalité) peut être stockée dans le cache et lors des exécutions suivantes, les données sont alors accessibles beaucoup plus rapidement. Lorsque les accès mémoire deviennent plus rapides alors ce sont les instructions de calcul qui limitent les performances et c'est à ce moment là que les instructions vectorielles sont les plus efficaces.

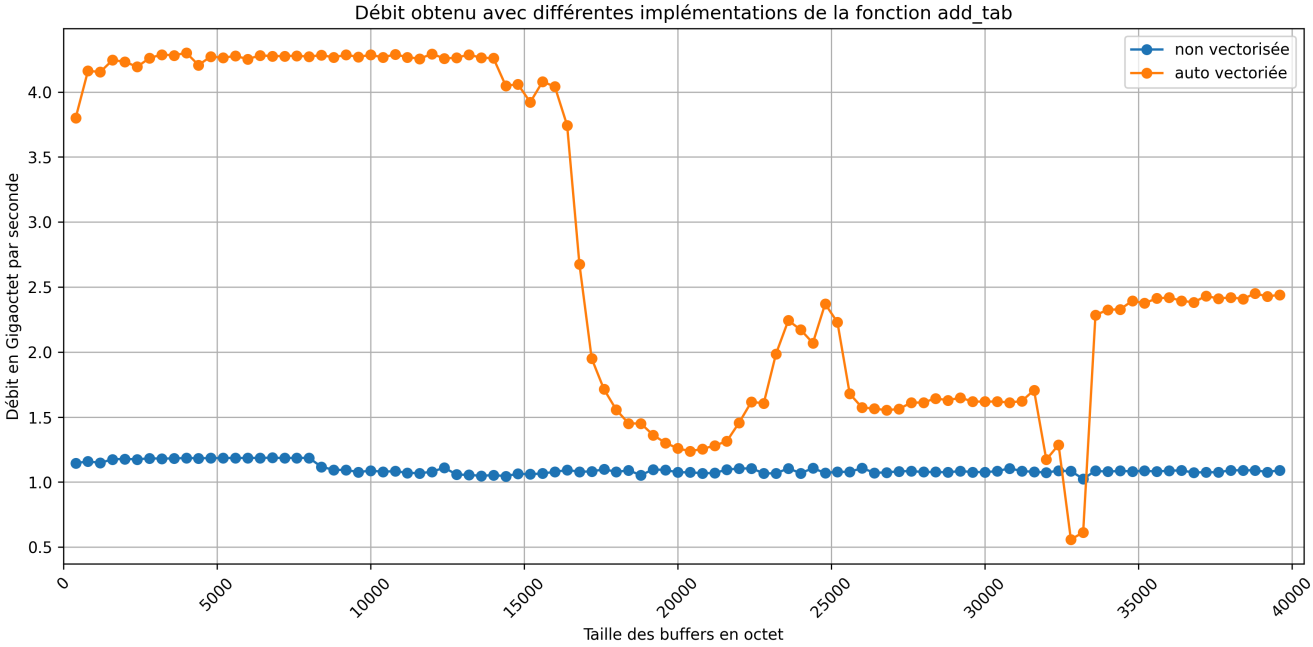

Fig. 9. – Graphique représentant les débits de données des versions auto-vectorisée et non vectorisée de la fonction `add_tab` selon la taille des tableaux d'entrée et de sortie.

10.1.3. Version intrinsics RVV et Version MIPP

Intéressons nous maintenant aux versions vectorisées « à la main » à l'aide des fonctions intrinsèques et du wrapper MIPP.

Le Tableau 5 nous montre les codes des deux versions. Dans celles-ci comme dans les versions précédentes on retrouve les instructions de lecture/écriture mémoire et d'addition. On remarque cependant quelques différences par rapport à la fonction auto vectorisée.

La première est que l'instruction `vsetvl` est maintenant en dehors de la boucle. La raison est que, comme spécifié dans les Chapitre 4.1 et Chapitre 8.1, MIPP a besoin de vecteur de taille fixe, la taille doit donc être déterminée au début de la fonction et pour toute l'exécution. Cette contrainte oblige à traiter le cas particulier des derniers éléments à calculer qui peuvent ne pas remplir complètement un vecteur. Pour régler ce problèmes deux solutions s'offrent à nous, la première est de masquer le vecteur afin que les cases de celui-ci qui ne correspondent pas à des données qui nous intéressent ne soient pas traitées. Une seconde solution, celle que nous avons choisi ici est une boucle de fin, séparée de la boucle principale, qui fait les calculs de manière scalaire pour les derniers éléments. Cette boucle de fin n'est pas visible dans le code Tableau 5 mais elle est bien présente dans le code avec lequel les évaluations de performances ont été effectués.

Une deuxième différence par rapport à la version vectorisée par le compilateur est que nos deux versions ci-dessous contiennent moins d'instructions au total dans la boucle principale, ceci est directement lié à l'instruction `vsetvl` qui est maintenant en dehors de la boucle. La version du compilateur a une boucle de 11 instructions au total, dont 36% d'instructions vectorielles, alors que

les versions intrinsèques et MIPP en ont respectivement 9 et 8 au total, avec 44% et 50% d'instructions vectorielles.

Dans la colonne de droite du Tableau 5 on remarque également que l'instruction d'addition est `vfadd` qui correspond à une addition sur des variables à virgule flottante alors que la fonction `add_tab` fait des additions sur des entiers. Ceci est dû à un choix d'implémentation de MIPP sur RISC-V qui utilise des vecteurs de float pour tous ses types de registres logiciels même pour les entiers. Ce choix a permis de simplifier l'implémentation.

```asm
add_tab_intr_rvv:
  [...]
  vsetvli a5,a0,e32,m1,ta,ma
  slli  a4,a5,2
  [...]
.L63:
  vle32.v v1,0(a1)
  vle32.v v2,0(a2)
  addw  a6,a6,t1
  add a1,a1,a4
  add a2,a2,a4
  vadd.vv v1,v1,v2
  vse32.v v1,0(a3)
  add a3,a3,a4
  bltu  a6,a7,.L63
  [...]
```

```asm
add_tab_mipp_rvv:
  [...]
  vsetvli zero,a5,e32,m1,ta,ma
.L16:
  vle32.v v1,0(a6)
  vle32.v v2,0(a7)
  addi  a6,a6,32
  addi  a7,a7,32
  vfadd.vv  v1,v1,v2
  vse32.v v1,0(a4)
  addi  a4,a4,32
  bne a6,t3,.L16
  [...]
```

Tableau 5. – Code assembleur RISC-V des versions intrinsèques et MIPP de la fonction `add_tab`

La Fig. 10 est similaire à la Fig. 9, en plus des performances des versions auto-vectorisée et non vectorisée comme vu plus haut, on y retrouve les performances de la version intrinsèque en vert et de la version MIPP en rouge.

On observe que les versions intrinsèques et MIPP ont des performances similaires avec des débits au dessus 4.5 Go/s pour des tableaux inférieurs à 16 Koctet, ce qui nous fait un gain de 9% par rapport à la version auto vectorisée. Ce gain est dû à un nombre plus réduit d'instructions exécutées dans la boucle principale du fait du déplacement hors de la boucle de l'instruction `vsetvl` comme vu précédemment.

Comme pour la version vectorisée par le compilateur, les performances se situent entre 0.5 et 2.5 Go/s (entre 1 et 2.5 Go/s pour la version intrinsèque) pour des tableaux de taille supérieure à 16 Koctet.

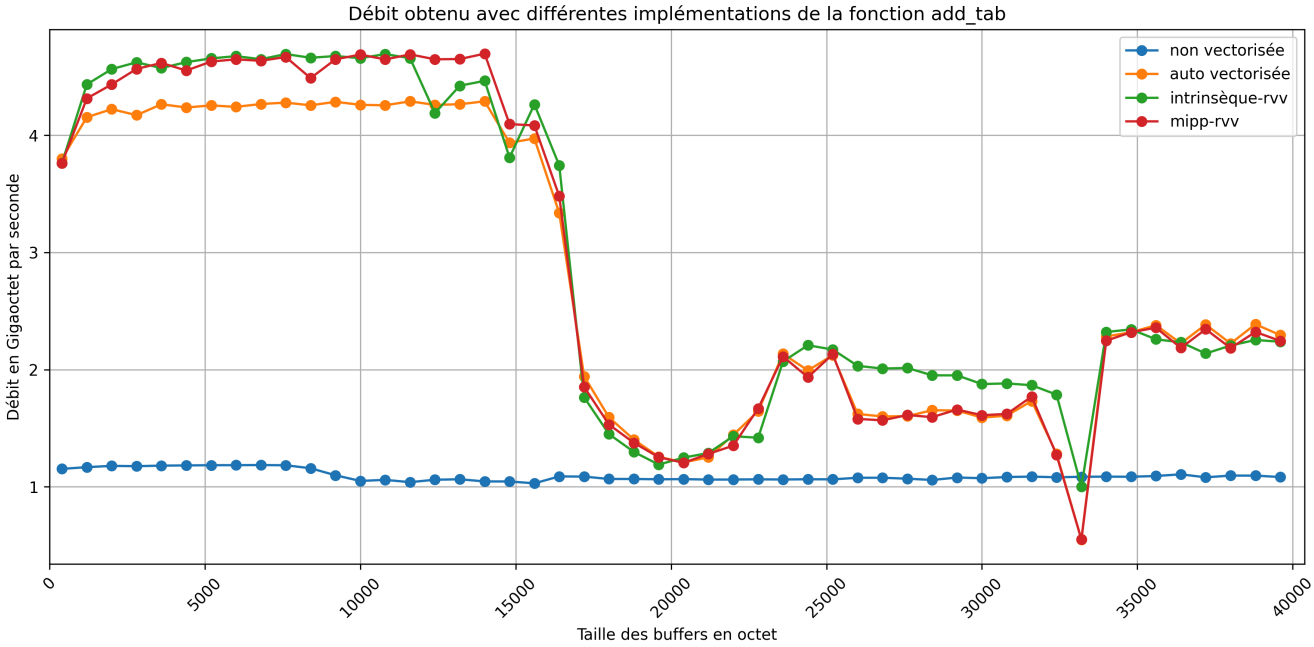

Fig. 10. – Graphique représentant les débits de données des versions scalaire auto-vectorisée, non vectorisé, intrinsèque et mipp de la fonction `add_tab`.

Dans cette section, j'ai pu montrer la capacité du compilateur à vectoriser une fonction d'addition très simple avec un gain de performance de près de 400% par rapport à une version non vectorisée. Malgré cette efficacité du compilateur, on a pu voir également l'intérêt des fonctions intrinsèques et de MIPP dans ce cas très simple avec un gain de 9% par rapport à la version auto vectorisée.

Ces premiers résultats encourageants montrent que la vectorisation à la main avec les intrinsèques ou avec MIPP est meilleure que l'auto-vectorisation par le compilateur et que les performances de MIPP sont similaires à celles des intrinsèques.

10.1.4. Le paramètre LMUL

Comme vu précédemment, une des particularités de RVV est la possibilité de fusionner plusieurs registres vectoriels pour en former un plus gros en faisant varier le paramètre LMUL. Afin de voir si les performances varient en fonction de ce paramètre, j'ai testé les performances de la fonction `add_tab` avec différentes valeurs de LMUL. Pour cela j'ai repris la version intrinsèque de la fonction `add_tab` et au lieu d'utiliser des types `vint32m1_t`, j'ai utilisé des types `vint32mX_t` avec X prenant les valeurs 1, 2, 4 et 8.

Au niveau du code assembleur, changer la valeur de LMUL implique simplement de spécifier la valeur du paramètre lors de l'appel à l'instruction `vsetvl` (lignes 3 dans le Tableau 6) et de n'utiliser que les registres dont le numéro est un multiple de la valeur de LMUL (exemple si LMUL=4, les registres utilisables sont v0, v4, v8, ...).

```asm
add_tab_intr_rvv:
    [...]
    vsetvli a5,a0,e32,m1,ta,ma
    slli  a4,a5,2
    [...]
.L63:
    vle32.v v1,0(a1)
    vle32.v v2,0(a2)
    addw  a6,a6,t1
    add a1,a1,a4
    add a2,a2,a4
    vadd.vv v1,v1,v2
    vse32.v v1,0(a3)
    add a3,a3,a4
    bltu  a6,a7,.L63
    [...]
```

```asm
add_tab_intr_rvv_m4:
    [...]
    vsetvli a5,a0,e32,m4,ta,ma
    slli  a4,a5,2
    [...]
.L39:
    vle32.v v4,0(a1)
    vle32.v v8,0(a2)
    addw  a6,a6,t1
    add a1,a1,a4
    add a2,a2,a4
    vadd.vv v4,v4,v8
    vse32.v v4,0(a3)
    add a3,a3,a4
    bltu  a6,a7,.L39
    [...]
```

Tableau 6. – Code assembleur des versions intrinsèques de la fonction `add_tab` avec le paramètre LMUL=1 à gauche et LMUL=4 à droite

Le graphique Fig. 11 donne les performances de la version intrinsèque de `add_tab` en fonction de la valeur du paramètre LMUL. Ici on s'intéressera surtout aux performances pour des tableaux de taille inférieure à 16 Ko car pour des tableaux plus grands les différences de performance sont moindres. On observe que les performances sont encore meilleures avec des valeurs de LMUL supérieurs à 1. La courbe de la version LMUL=2 (en jaune) oscille autour des 5.2 Go/s avec un pique à 5.7 Go/s, ce qui représente un gain de 10% par rapport à la version LMUL=1. Les versions LMUL=4 et LMUL=8 fluctuent autours des 6 Go/s avec un pique de 7 Go/s, ce qui fait un gain de 27% par rapport à la version LMUL=1.

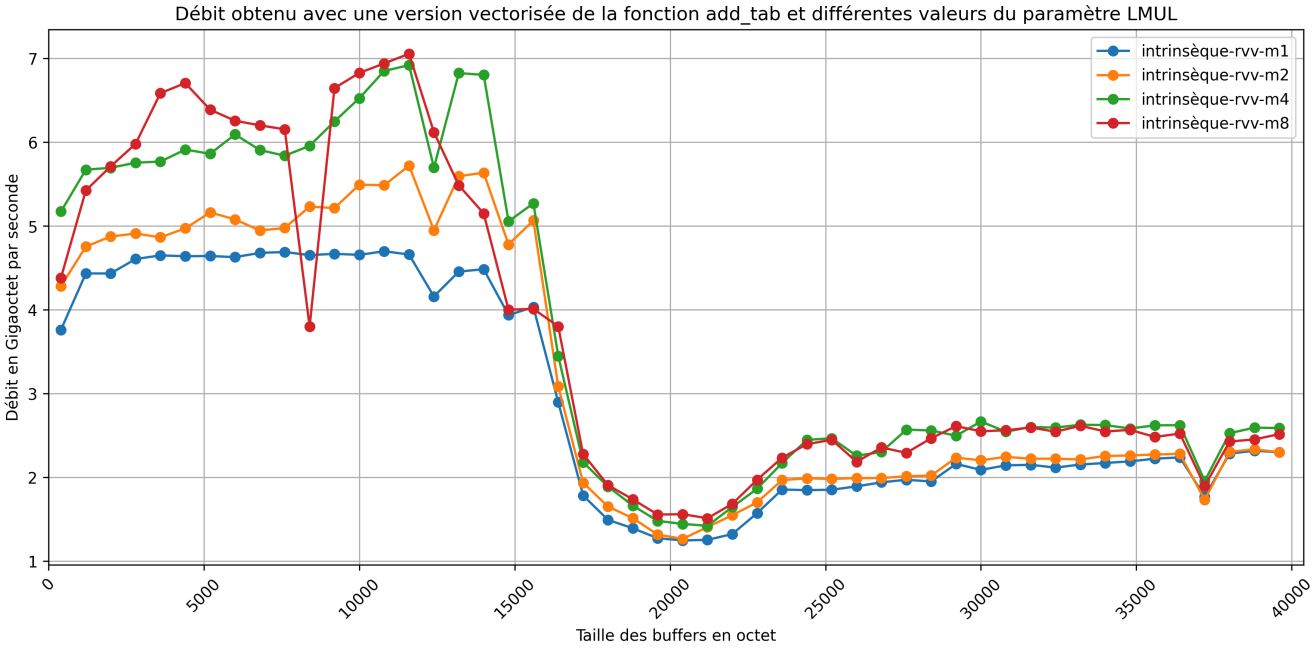

Fig. 11. – Graphique représentant les débits de données de la version intrinsèques de la fonction `add_tab` et différentes valeurs du paramètres LMUL

On remarque donc que le paramètre LMUL peut jouer un rôle important pour gagner en performance, dans notre exemple on a réussi à gagner près de 30% de performance. L'inconvénient d'augmenter la valeur de LMUL est qu'on divise le nombre de registres vectoriels utilisables : cette contrainte ne pose pas de problème dans notre exemple très simple mais peut drastiquement faire baisser les performances d'applications plus complexes qui nécessitent un grand nombre de registres.

On a montré qu'avec les instructions vectorielles, la fonction `add_tab` pouvait être accélérée de 400% par rapport à la version scalaire et de 9% par rapport à la version auto-vectorisé par le compilateur.

Étant donné que les registres vectoriels du matériel utilisé sont de 256 bits, soit 8 entiers de 32 bits, on peut se demander pourquoi les performances des versions vectorisées ne sont pas 8 fois supérieures aux performances de la version scalaire non vectorisée. Cette différence peut s'expliquer avec les accès mémoire, comme on peut le voir dans le Tableau 5, les instructions mémoire représente 35% du code assembleur et même lorsque les données sont dans le cache, faire un accès mémoire est généralement plus coûteux que de faire une instruction de calcul.

10.2. Mandelbrot

10.2.1. Validation fonctionnelle

Pour la validation fonctionnelle des différentes versions de l'algorithme de mandelbrot, je construis une image à partir des données de sortie. L'image Fig. 12 correspond à l'image de la version scalaire qui me sert de référence pour valider le bon comportement des autres versions. Pour qu'une nouvelle implémentation de l'algorithme de mandelbrot soit validé, son image de sortie doit être strictement identique à l'image Fig. 12.

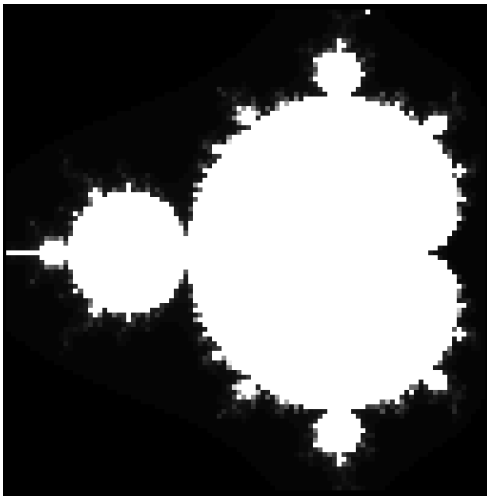

Fig. 12. – Image construite à partir des données de sortie de la fonction mandelbrot

10.2.2. Premières versions vectorielles

Le compilateur étant incapable de vectoriser automatiquement l'algorithme de Mandelbrot, nous nous intéresserons, ici, aux versions scalaire, vectorisée avec les intrinsèques et vectorisée avec MIPP.

Les premières versions de la fonction `mandelbrot` correspondent, ici, à la première implémentation avec les intrinsèques et à la première implémentation avec MIPP (et avec LMUL=1) que j'ai pu obtenir et qui étaient valides d'un point de vu fonctionnelle (génèrent des images identiques à la Fig. 12). Nous verrons ensuite comment cette première version peut être améliorée pour obtenir de meilleures performances.

Le Tableau 7 contient le code assembleur correspondant à différentes versions de la fonction `mandelbrot`. Pour que le code reste assez lisible, le Tableau 7 ne contient que la boucle `while` de la fonction (ligne 9 à 17 du code C -> Liste 2). Dans la colonne de gauche, on retrouve la version scalaire avec les instructions scalaires sur les `float` (`fsub`, `fadd`, `fmadd`, ...). Dans la colonne milieu, on retrouve la version intrinsèque, on peut voir les instructions vectorielles sur les `float` (`vfsub`, `vfadd`, `vfmacc`, ...), on remarque que cette version nécessite 17 instructions au total (pour la boucle `while`) contre 11 pour la version scalaire, avec 13 instructions vectorielles soit 76% d'instructions vectorielles. Dans la colonne de droite, on peut voir enfin la version MIPP avec 18 instructions au total et 15 instructions vectorielles soit 83% d'instructions vectorielles.

```asm
1  mandelbrot_scal:
2    [...]
3  .L153:
4    beq a4,a5,.L154
5    fsub.s  fa4,fa4,fa2
6    fadd.s  fa2,fa5,fa5
7    addi  a5,a5,1
8    fadd.s  fa5,fa4,fa1
9    fmadd.s
     fa3,fa2,fa3,ft0
10   fmul.s  fa4,fa5,fa5
11   fmul.s  fa2,fa3,fa3
12   fadd.s  fa0,fa4,fa2
13   fle.s a3,fa0,ft1
14   bne a3,zero,.L153
15 .L154:
16   sw  a5,0(a2)
17   [...]
```

```asm
1  mandelbrot_intr:
2    [...]
3  .L340:
4    blt a2,zero,.L313
5  .L314:
6    vfadd.vv  v0,v2,v4
7    vfadd.vv  v1,v1,v1
8    vfsub.vv  v2,v2,v4
9    vmv1r.v v4,v3
10   vmflt.vv  v0,v0,v9
11   addi  a4,a4,1
12   vmv1r.v v3,v6
13   vfmacc.vv v3,v1,v4
14   vfadd.vv  v1,v2,v7
15   vmerge.vim
     v4,v8,1,v0
16   vfirst.m  a2,v0
17   vadd.vv v5,v5,v4
18   vfmul.vv  v2,v1,v1
19   vfmul.vv  v4,v3,v3
20   bgtu  a3,a4,.L340
21 .L313:
22   vse32.v v5,0(a0)
23   [...]
```

```asm
1  mandelbrot_mipp:
2    [...]
3  .L248:
4    vfadd.vv  v0,v5,v3
5    vfadd.vv  v1,v1,v1
6    vmv1r.v v2,v4
7    vfsub.vv  v3,v3,v5
8    vmflt.vv  v0,v0,v10
9    vmv1r.v v4,v7
10   vfmacc.vv v4,v1,v2
11   addi  a4,a4,1
12   vmerge.vim
     v2,v9,1,v0
13   vfadd.vv  v1,v8,v3
14   vfirst.m  a0,v0
15   vfcvt.f.x.v v2,v2
16   vfmul.vv  v5,v4,v4
17   vfmul.vv  v3,v1,v1
18   vfcvt.x.f.v v2,v2
19   vfadd.vv  v6,v6,v2
20   bleu  a1,a4,.L247
21   bge a0,zero,.L248
22 .L247:
23   vse32.v v6,0(a3)
24   [...]
```

Tableau 7. – Code assembleur des versions scalaire, intrinsèque et mipp de la fonction mandelbrot

Dans le Tableau 7, on observe également, à la ligne 15 du code de la version intrinsèque et à la ligne 12 du code de la version MIPP, l'instruction `vmerge`, cette instruction permet de fusionner un vecteur et un scalaire avec un masque, comme on peut le voir sur la Fig. 13.

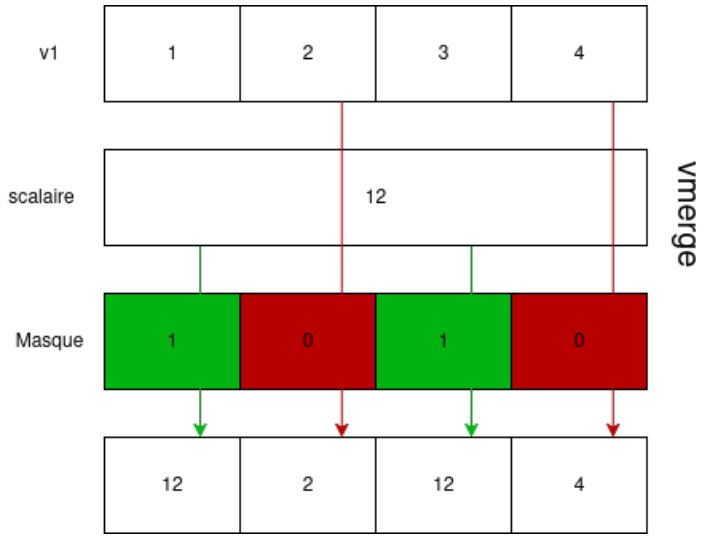

Fig. 13. – Schéma de l'opération effectuée par l'instruction `vmerge`

Le graphique de la Fig. 14 correspond au speedup des versions vectorielles de la fonction `mandelbrot` par rapport à la version scalaire. Autrement dit les valeurs correspondant aux versions vectorielles indiquent combien de fois elles sont plus rapides que la version scalaire. En rouge on peut voir le speedup de la version scalaire qui sert de référence, sa valeur est donc à 1. Ensuite on observe que les versions intrinsèque (en vert) et MIPP (en bleu) sont respectivement 6.7 et 6 fois plus rapides que la version scalaire. Le gain de performances est nettement plus important que ce qu'on a pu voir avec la fonction `add_tab` dont les versions vectorielles n'amélioraient les performances que d'un facteur 4 maximum. Cette différence s'explique par le fait que la fonction `mandelbrot` est presque exclusivement calculatoire (aucune instruction mémoire dans la boucle `while`).

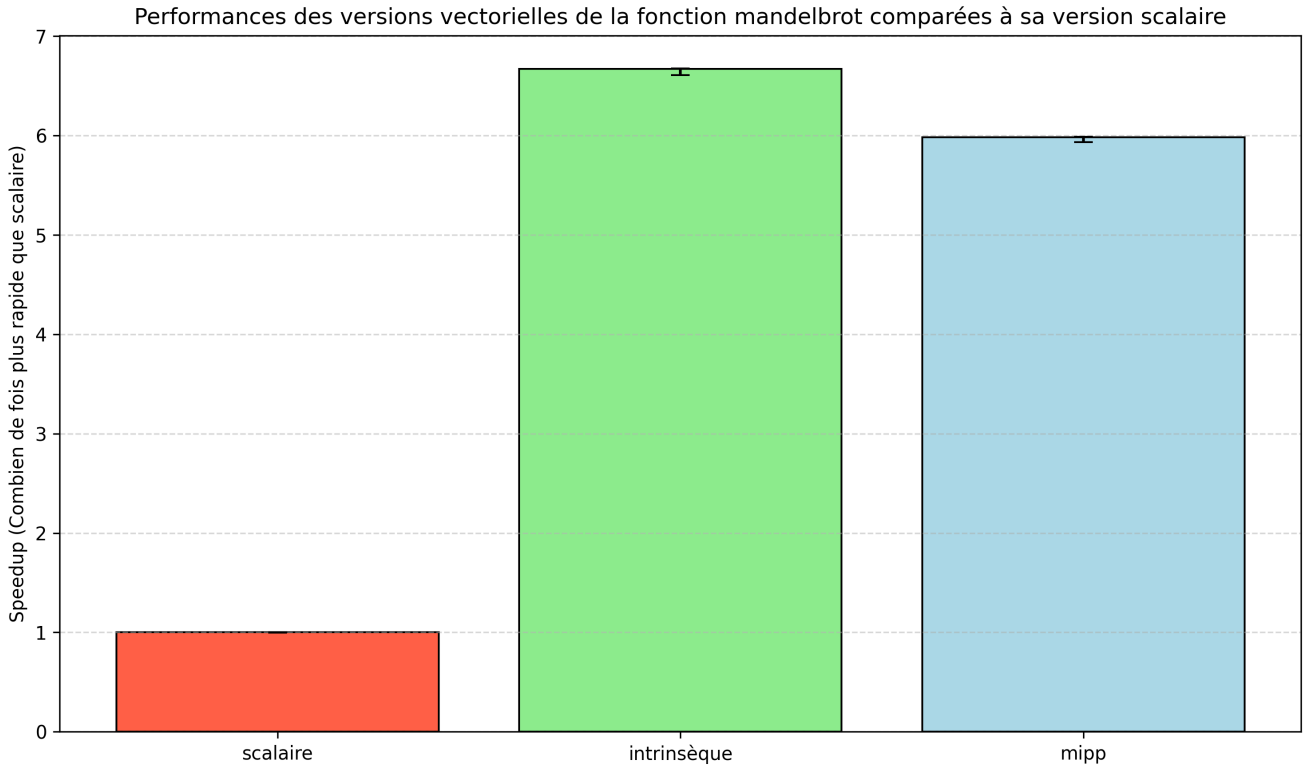

Fig. 14. – Diagramme en barres montrant les performances des versions intrinsèques et MIPP de mandelbrot (avec LMUL=1) comparé à la version scalaire

En observant la Fig. 14, on peut tout de même se demander pour quelle raison les performances des versions intrinsèque et MIPP sont différentes et est ce que la version MIPP ne pourrait pas être améliorée. Quand on regarde de plus près le code assembleur de la version MIPP (Tableau 7, colonne de droite), on remarque qu'il y a deux instructions de conversion successives (ligne 15 et 18). On remarque également que, bizarrement, ces instructions convertissent la donnée d'un même registre de `foat` vers `int` puis de `int` vers `float` sans que la donnée soit utilisé entre les deux, ces conversions ne servent donc à rien. Pour comprendre pourquoi ces instructions de conversions sont là, il faut regarder au niveau du code C++ de la version MIPP de la fonction `mandelbrot`. Le code Liste 3 nous montre la boucle `while` de la version MIPP. On peut y voir, à la ligne 5, une conversion de `float` vers `unsigned int`, c'est cette ligne qui nous intéresse. Au moment d'incrémenter les éléments du vecteur `viter`, il est nécessaire de convertir un masque dans le type correspondant aux éléments de `viter`, ici des entiers non signés. Dans un premier temps la méthode `mipp::toReg` ne me permettait de convertir un masque que vers un vecteur de `float`, pour cette première version, j'ai simplement reconverti le vecteur de `float` en vecteur d'entiers à l'aide de la méthode `mipp::cvt`.

```C
1  void mandelbrot_mipp(){
2    [...]
3    do{
4      mask = mipp::cmplt<float>((zx2+zy2), v4);
5      viter = viter + mipp::cvt<float, unsigned>(mipp::toReg<float>(mask));
6      zy = mipp::fmadd(zx+zx, zy, cy);
7      zx = (zx2-zy2)+cx;
8      zx2 = zx * zx;
9      zy2 = zy * zy;
10     ++iter;
11   } while (iter < maxIter && mipp::testz(mask));
12   viter.store(res);
13   [...]
```

Liste 3. – Noyau de la version MIPP de l'algorithme de Mandelbrot

On constate donc que cette première version MIPP est assez naïve car elle ne cherche pas être la plus efficace possible, elle est également limité par le fait que le wrapper MIPP n'est encore implémenté que partiellement pour RVV (il manque la méthode `mipp::toReg<unsigned>`). Malgré tout cette version montre des performances tout à fait correct par rapport à la version intrinsèque. Voyons maintenant si nous pouvons proposer une meilleure implémentation de Mandelbrot avec MIPP.

10.2.3. Deuxième version

Cette deuxième version a pour objectif d'améliorer les performances de la version MIPP pour se rapprocher de celle de la version intrinsèque.

À première vue, l'approche la plus simple serait de construire la méthode `mipp::toReg<unsigned>()` permettant de changer la ligne 5 Liste 2 pour ne plus avoir à faire de conversion du tout. Pour des raisons encore obscures, je n'ai pas réussi à faire en sorte que la méthode `mipp::toReg<unsigned>()` soit fonctionnelle. Finalement j'ai décidé d'éviter le problème et d'assumer que le vecteur `viter` contiendrait des éléments de type `float`. Cette simplification permet de déplacer la conversion en dehors de la boucle `while` au moment de l'écriture du résultat en mémoire. Comme on peut le voir Liste 4, la conversion se fait maintenant en dehors de la boucle (ligne 10).

```c
do{
  mask = mipp::cmplt<float>((zx2+zy2), v4);
  viter = viter + mipp::toReg<float>(mask);
  zy = mipp::fmadd(zx+zx, zy, cy);
  zx = (zx2-zy2)+cx;
  zx2 = zx * zx;
  zy2 = zy * zy;
  ++iter;
} while (iter < maxIter && mipp::testz(mask));
mipp::cvt<float,unsigned>(viter).store(res);
```

Liste 4. – Nouveau noyau de la version MIPP de Mandelbrot

Dans le code assembleur généré à partir de la nouvelle version, que l'on peut voir Tableau 8 colonne de gauche, la conversion `float` vers `int` à bien été déplacée en dehors de la boucle à la ligne 22 juste avant l'écriture en mémoire de la donnée (ligne 24). En revanche on remarque qu'il y a toujours la conversion `int` vers `float` (ligne 15), ceci est dû au fait que l'instruction `vmerge` (ligne 12) génère des valeurs entières. Ces valeurs, correspondant au registre `v5` à partir de la ligne 12, ce sont les incréments pour le vecteur `viter` or nous avons dit plus haut que nous acceptions que `viter` soit un vecteur de `float` stockant des entiers, le registre `v5` doit donc être converti de `int` vers `float`.

```asm
mandelbrot_mipp:
  [...]
.L407:
  vfadd.vv  v0,v6,v2
  vfadd.vv  v1,v1,v1
  vmv1r.v v5,v4
  vfsub.vv  v2,v2,v6
  vmflt.vv  v0,v0,v10
  vmv1r.v v4,v7
  vfmacc.vv v4,v1,v5
  addi  a4,a4,1
  vmerge.vim  v5,v9,1,v0
  vfadd.vv  v1,v8,v2
  vfirst.m  a0,v0
  vfcvt.f.x.v v5,v5
  vfmul.vv  v6,v4,v4
  vfmul.vv  v2,v1,v1
  vfadd.vv  v3,v5,v3
  bleu  a1,a4,.L406
  bge a0,zero,.L407
.L406:
  vfcvt.x.f.v v3,v3
  [...]
  vse32.v  v3,0(a3)
  [...]
```

```asm
mandelbrot_intr:
  [...]
.L340:
  blt a2,zero,.L313
.L314:
  vfadd.vv  v0,v2,v4
  vfadd.vv  v1,v1,v1
  vfsub.vv  v2,v2,v4
  vmv1r.v v4,v3
  vmflt.vv  v0,v0,v9
  addi  a4,a4,1
  vmv1r.v v3,v6
  vfmacc.vv v3,v1,v4
  vfadd.vv  v1,v2,v7
  vmerge.vim  v4,v8,1,v0
  vfirst.m  a2,v0
  vadd.vv v5,v5,v4
  vfmul.vv  v2,v1,v1
  vfmul.vv  v4,v3,v3
  bgtu  a3,a4,.L340
.L313:
  vse32.v v5,0(a0)
  [...]
```

Tableau 8. – Code assembleur de la version MIPP de la fonction mandelbrot

La Fig. 15 nous montre les performances de la seconde version de Mandelbrot avec MIPP (courbe en bleu). On peut voir que le speedup de la nouvelle version MIPP est nettement meilleur avec une accélération de 6.9 par rapport à la version scalaire contre 6.7 avec la première version. On remarque également que notre version MIPP dépasse légèrement la version intrinsèque qui est à 6.7 par rapport à la version scalaire, ceci est probablement dû à l'ordre dans lesquels les instructions sont exécutées car comme on peut le voir dans le code assembleur Tableau 8, les instructions de la boucle `while` sont pratiquement les mêmes (mis à par l'instruction de conversion et les branchements qui sont un peu différents).

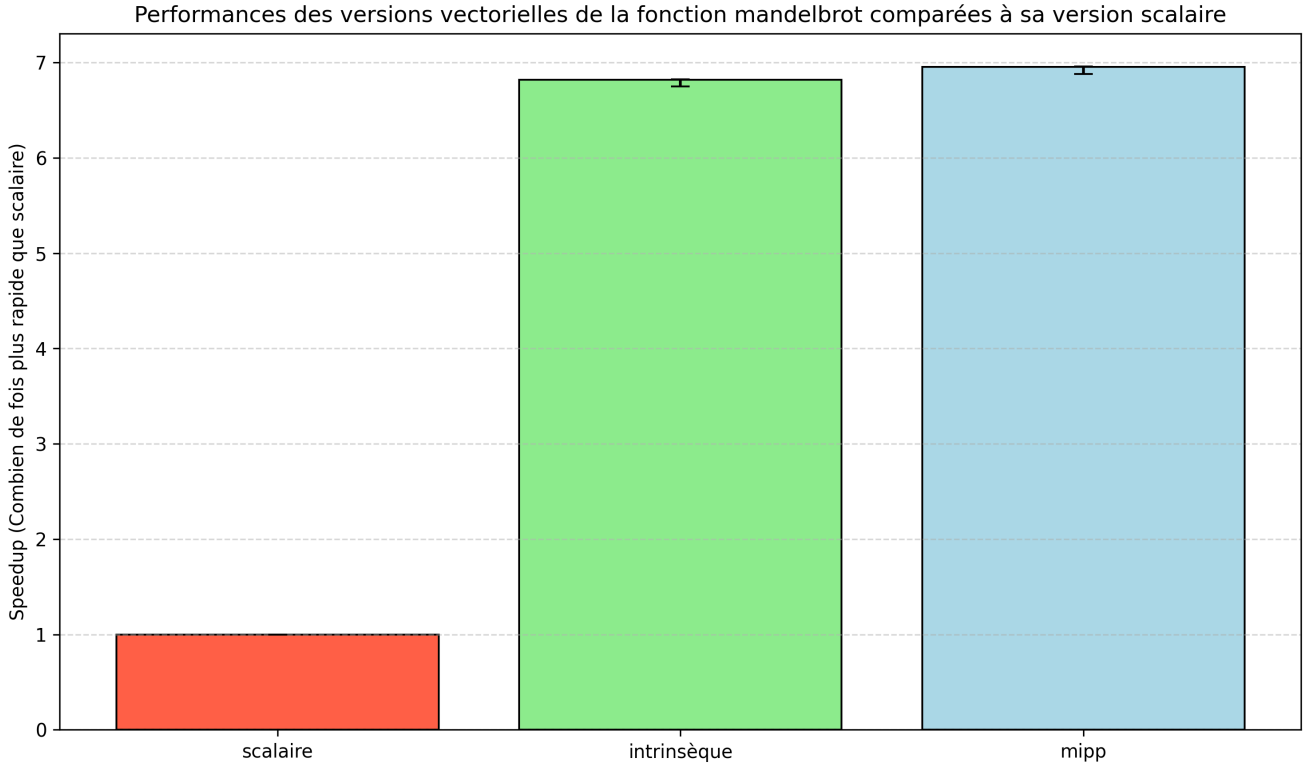

Fig. 15. – Diagramme en barres montrant les performances de la seconde version de l'implémentation MIPP de mandelbrot comparées à la version scalaire

10.2.4. Mandelbrot avec LMUL > 1

Comme pour la fonction `add_tab`, on va s'intéresser aux performances des versions vectorielles de l'algorithme de Mandelbrot avec différentes valeurs du paramère LMUL. Contrairement aux mesures de `add_tab` qu'on a pu voir dans le Chapitre 10.1.4, nous allons également tester les performances de MIPP, en plus de celles des intrinsèques, avec différentes valeurs de LMUL et voir si on peut encore obtenir de meilleures des performances.

Pour pouvoir utiliser MIPP avec la valeur de LMUL voulue, j'ai modifié la partie RVV de MIPP pour que le code s'adapte en fonction d'un paramètre choisi à la compilation. Il est donc nécessaire de recompiler MIPP pour changer le paramètre LMUL.

Pour la version intrinsèque, comme pour le Chapitre 10.1.4, on adapte les types et les fonctions intrinsèques à la valeur du paramètre LMUL qui nous intéresse.

Une fois MIPP recompilé et la version intrinsèque adaptée pour des valeurs de LMUL supérieur à 1, on obtient les performances que l'on peut voir sur le diagramme en barres Fig. 16. À gauche du graphique, comme pour la Fig. 15, on retrouve les performances des versions intrinsèque et MIPP avec LMUL=1. En rouge, la version intrinsèque avec LMUL=1 sert de référence pour les autres

versions. Au milieu du diagramme, les barres bleu et verte correspondent aux versions intrinsèque et MIPP avec LMUL=2, elles sont, respectivement, 29% et 17% plus rapides que la version de référence. Enfin à droite se trouve les versions avec LMUL=4 qui sont 12% plus rapide pour la version intrinsèque et 8% plus rapide pour la version MIPP.

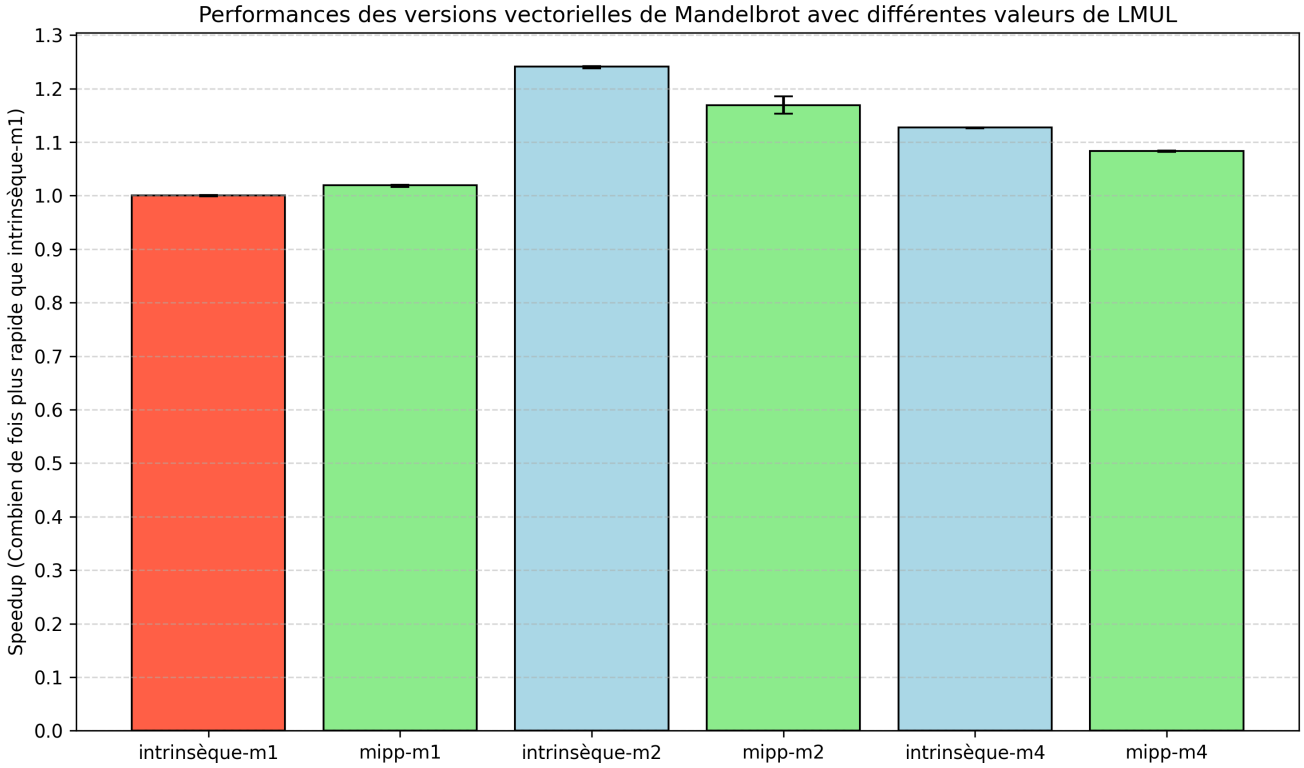

Fig. 16. – Diagramme en barres montrant les performances des versions vectorielles de Mandelbrot avec différentes valeurs de LMUL

On constate donc qu'avec des valeurs de LMUL supérieures à 1, l'algorithme de Mandelbrot gagne encore en performance (entre 10% et 30% environ par rapport aux versions LMUL=1). Ce gain est assez similaire à celui que nous avions obtenu pour la fonction `add_tab` (voir Chapitre 10.1.4). On constate tout de même que cette fois-ci, les performances sont moindres avec LMUL=4 qu'avec LMUL=2, ce qui laisse penser que pour des valeurs de LMUL supérieure à 2 le banc de registres est saturé. Et effectivement, comme on peut le voir dans le code assembleur Liste 5, la version LMUL=4 de la fonction `mandelbrot` (colonne de droite) a besoin de faire un accès mémoire (`vl4re32.v`, ligne 14), pour ramener dans le registre `v4` une donnée qui est directement dans en registre dans les autres versions (correspond au registre `v8` dans la version LMUL=2, voir Tableau 9, colonne de gauche, ligne 17). Cette donnée correspond à un vecteur dont tous les éléments ont une valeur de 4, elle est nécessaire pour vérifier la condition d'arrêt de la boucle. Remarque : comme cette valeur de 4 n'est jamais modifiée, pas besoin de réécrire la donnée en mémoire.

```asm
    [...]
    vsetvli a5,zero,e32,m4,ta,ma
    [...]
  .L421:
    blt a6,zero,.L393
  .L394:
    vfadd.vv  v0,v28,v8
    vfsub.vv  v8,v28,v8
    vl4re32.v v28,0(sp)
    vfadd.vv  v4,v4,v4
    addi  a6,sp,128
    addi  a4,a4,1
    vfmacc.vv v28,v4,v12
    vl4re32.v v4,0(a6)
    vmv4r.v v12,v28
    vmflt.vv  v0,v0,v4
    vfadd.vv  v4,v8,v24
    vmerge.vim  v8,v20,1,v0
    vfirst.m  a6,v0
    vfmul.vv  v28,v4,v4
    vadd.vv v16,v16,v8
    vfmul.vv  v8,v12,v12
    bgtu  a3,a4,.L421
  .L393:
    vse32.v v16,0(a0)
    [...]
```

Liste 5. – Code assembleur de la version intrinsèque de la fonction `mandelbrot` avec LMUL=4

Dans le Tableau 9, on retrouve les codes assembleurs des versions intrinsèque et MIPP avec LMUL=2. Dans la Fig. 16, on peut voir qu'il y a une différence de performance entre la version intrinsèque avec LMUL=2 et la version MIPP avec LMUL=2, cette différence est de 5%, elle peut s'expliquer par le fait que la boucle `while` de la version intrinsèque (colonne de gauche du Tableau 9) contient une instruction de moins (la ligne 7 ne compte pas comme une instruction) que la version MIPP (colonnde droite du Tableau 9).

```asm
mandelbrot_intrinsics_m2:
  [...]
  vsetvli a5,zero,e32,m2,ta,ma
  [...]
.L380:
  blt a2,zero,.L353
.L354:
  vfadd.vv  v0,v4,v8
  vfadd.vv  v2,v2,v2
  vfsub.vv  v4,v4,v8
  vmv2r.v v8,v6
  vmflt.vv  v0,v0,v18
  addi  a4,a4,1
  vmv2r.v v6,v12
  vfmacc.vv v6,v2,v8
  vfadd.vv  v2,v4,v14
  vmerge.vim  v8,v16,1,v0
  vfirst.m  a2,v0
  vadd.vv v10,v10,v8
  vfmul.vv  v4,v2,v2
  vfmul.vv  v8,v6,v6
  bgtu  a3,a4,.L380
.L353:
  vse32.v v10,0(a0)
 [...]
```

```asm
mandelbrot_mipp_m2:
  [...]
  vsetvli a5,zero,e32,m2,ta,ma
  [...]
.L407:
  vfadd.vv  v0,v30,v4
  vfadd.vv  v2,v2,v2
  vmv2r.v v28,v8
  vfsub.vv  v4,v4,v30
  vmflt.vv  v0,v0,v14
  vmv2r.v v8,v10
  vfmacc.vv v8,v2,v28
  addi  a4,a4,1
  vmerge.vim  v28,v12,1,v0
  vfadd.vv  v2,v26,v4
  vfirst.m  a0,v0
  vfcvt.f.x.v v28,v28
  vfmul.vv  v30,v8,v8
  vfmul.vv  v4,v2,v2
  vfadd.vv  v6,v28,v6
  bleu  a1,a4,.L406
  bge a0,zero,.L407
.L406:
  vfcvt.x.f.v v6,v6
 [...]
  vse32.v v6,0(a3)
 [...]
```

Tableau 9. – Code assembleur des versions intrinsèques et MIPP avec LMUL=2 de la fonction `mandelbrot`

11. Récapitulatif de la procédure de recette

Voyons maintenant si les objectifs fixés au début du stage peuvent être validés. Pour valider la première étape, chaque application implémentée avec MIPP devait passer un test fonctionnel pour s'assurer que MIPP est utilisable et que ses performances sont comparables avec d'autres implémentations. Ensuite, pour valider la deuxième étape, les performances de MIPP devait être supérieures aux versions auto-vectorisées (si elles existent) et du même ordre de grandeur que les versions vectorisées à la main avec les fonctions intrinsèques.

Pour l'addition vectorielle, comme expliqué dans le Chapitre 10.1, la version MIPP de la fonction `add_tab` a donné en sortie des valeurs strictement identiques aux valeurs de sortie de référence de la version scalaire. Ensuite on a pu voir que les performances de la version MIPP était meilleures que celles de la version auto-vectorisée par le compilateur (de 9%) et identique à la version intrinsèque. Enfin on a pu voir qu'avec un paramètre LMUL supérieur à 1, on pouvait obtenir de meilleures performances. On peut dire que pour cette application, les objectifs ont été atteint.

Pour l'algorithme de Mandelbrot, que nous avons vu dans le Chapitre 10.2, l'image sortie à partir de la version MIPP de l'algorithme est bien identique à l'image de la Fig. 12, ensuite le compilateur n'a pas été capable de donner une version auto-vectorisée, la version du compilateur n'a donc pas été pris en compte. Comme pour l'application `add_tab`, les performances de MIPP sont identiques aux performances de la version intrinsèque, et on a montré que l'on pouvait obtenir de meilleures performances avec des valeurs de LMUL supérieures à 1. Pour l'algorithme de Mandelbrot, on peut également dire que les objectifs ont été atteint.

12. Conclusion

L'objectif de ce stage était de porter le wrapper MIPP sur le jeu d'instructions vectorielles de RISC-V, et de montrer son intérêt par rapport à d'autres solutions comme les fonctions intrinsèques. Nous avons pu voir plus haut l'efficacité de MIPP sur une application simple, limitée par les accès mémoires, et sur une application plus complexe et plus calculatoire, dans les deux cas l'overhead de MIPP par rapport aux fonctions intrinsèques est, au pire des cas de l'ordre de 10% (avec une implémentation partielle de MIPP et quand le wrapper est utilisé naïvement sans plus d'optimisation), et dans le meilleur des cas on observe pas de différence entre une implémentation MIPP et une implémentation avec les intrinsèques. À mon sens cette, la partie de MIPP sur laquelle j'ai travaillé constitue une brique solide sur laquelle s'appuyer pour les futures évolutions du wrapper avec RISC-V. Il reste néanmoins des améliorations à faire pour que MIPP sur RISC-V soit utilisable dans d'autres cas que ceux que nous avons vu. Par exemple, pour continuer d'avancer, Il serait intéressant de choisir une application plus concrète comme par exemple une application de traitement du signal. Une autre contribution pourrait être également de tester MIPP sur d'autres implémentations de RVV avec des tailles de registres vectoriels différentes.

13. Bibliographie

- cpu clock rate : https://en.wikipedia.org/wiki/Clock_rate
- Loi de Moore : https://fr.wikipedia.org/wiki/Loi_de_Moore
- Parallel computing : https://en.wikipedia.org/wiki/Parallel_computing
- SIMD : https://en.wikipedia.org/wiki/Single_instruction,_multiple_data
- RISC-V : https://github.com/riscv/riscv-isa-manual/releases/
- RVV : https://github.com/riscvarchive/riscv-v-spec/releases
- MIPP : https://github.com/aff3ct/MIPP