RAPPORT DE STAGE TECHNICIEN

GRENOBLE INP – Esisar, UGA 2024

Théorie et pratique de la programmation vectorielle pour le traitement de flux de caractères

LCIS
Laboratoire de Conception
et d'Intégration des Systèmes

LCIS
50 rue Barthelemy de
Laffemas, CS 10054
26902 Valence Cedex 9

Verimag, Bâtiment IMAG
Université Grenoble Alpes
150 place du torrent
38401 Saint Martin d'Hères

Zoë Courvoisier-Clément

Dates du stage	10/06/2024 – 02/08/2024
Maître·sse·s de stage	Laure GONNORD (GRENOBLE INP/LCIS) Bruno FERRES (UGA/VERIMAG)
Tuteur ESISAR	Jean-Luc KONING

Table des matières

1. Introduction **1**
1.1. Le laboratoire d'accueil: LCIS 1
1.2. Contexte du stage 1
1.3. Objectifs du stage 1
2. La programmation vectorielle **1**
2.1. Concept de base 2
2.2. Programmation vectorielle sur x86 3
2.2.1. Le vecteur le plus basique : `int` 3
2.2.2. Vecteurs SSE et AVX 3
2.3. Programmation vectorielle sur RISC-V 4
3. L'environnement RISC-V et son outillage **6**
3.1. Exécution de programmes RVV 7
3.2. Outillage additionnel et tâches accomplies 8
4. Réécriture d'algorithmes vers l'extension vectorielle RISC-V **9**
4.1. Dans quel but ? 9
4.2. Un exemple pratique : `memchr` 9
4.2.1. Approche vectorielle et implémentation AVX 10
4.2.2. Implémentation en RVV 12
4.3. Un cas de réécriture plus difficile 13
4.4. Variantes et sélection d'instruction sous RVV 16
4.5. Résultats de cette réécriture 18
5. Mesures et performances **18**
5.1. Conditions expérimentales 19
5.2. Comparaison des implémentations scalaires et vectorielles 19
5.3. Comparaison de variantes vectorielles 20
6. Conclusion **21**
Bibliographie **22**
A. Annexes ..... **i**
A.1. Détails des macros utilisées dans le code RVV i
A.2. Détails des macros utilisées dans le code AVX2 ii
A.3. Implémentations RVV des algorithmes restants ii
A.3.1. Implémentation de `mask` ii
A.3.2. Implémentation de `memcmp` iii
A.4. Exemple d'exécution de différents algorithmes v
A.4.1. Exécution de l'implémentation AVX de `memseq` v

Remerciements

Durant ce stage, je fus entourée par l'agréable équipe du LCIS, et j'aimerais remercier chacun·e des membres pour leur accueil chaleureux ainsi que leur gentillesse générale. Discuter avec les différent·e·s chercheur·euse·s de leurs travaux actuels et passés était infiniment passionnant, et a définitivement développé ma curiosité dans quelques nouveaux domaines.

En particulier, je souhaiterais remercier Laure Gonnord pour sa personnalité bienveillante et énergétique, pour ses conseils utiles tout au long de ces 8 dernières semaines, et bien sûr pour son aide inestimable lors de l'écriture de ce rapport, une partie particulièrement difficile de ce stage. Il en va de même pour Bruno Ferres, mon deuxième maître de stage, travaillant à VERIMAG, qui était toujours très réactif, et qui m'a constamment aidé à instinctivement remettre en question mes approches et résultats. Iels furent chacun·e très faciles à approcher, tout en maintenant la rigueur et le scepticisme nécessaire à des maîtres·se·s de stages. Ce stage fut incroyablement agréable, et je ne peux exprimer à quel point je me sens privilégiée et reconnaissante d'avoir pu travailler sur ce projet, qui combine avec tant d'élégance mes intérêts en programmation vectorielle, compilateurs, conception de processeurs, et sémantique. Je penserai certainement encore à ce stage et au projet **SxC** pendant un petit bout de temps.

J'aimerais aussi adresser ces remerciements à Sébastien Michelland et Arthur Baudet, qui m'ont perpétuellement mise au défi et encouragée à chaque étape de ce projet; ils m'ont aidé à réfléchir à bien des choses, liées ou non à ce stage, and furent constamment curieux chaque fois que je rencontrais un nouveau bug ou obstacle.

1. Introduction

1.1. Le laboratoire d'accueil: LCIS

J'ai effectué mon stage dans l'enceinte du LCIS, un laboratoire de l'Université Grenoble Alpes et de Grenoble-INP, dont les locaux se situent sur le campus UGA Valence. Dédié à la recherche publique, il est composé de plus de 60 enseignant·e·s chercheur·se·s, répartis entre trois équipes différentes (ORSYS, CO4SYS, CTSYS), spécialisées respectivement en télécommunications, en systèmes multi-agents, et en sécurité des systèmes distribués et embarqués. Ce stage s'est déroulé au sein de l'équipe CTSYS, avec Laure Gonnord comme maîtresse de stage.

1.2. Contexte du stage

« De Shannon à Cray » (**SxC**) est un projet de recherche collaborative accepté par l'ANR en juillet 2024, qui a pour but de développer un langage haut-niveau facilitant l'écriture de programmes de traitement de flux de caractères (par ex. pour la bio-informatique), tout en accélérant ces mêmes programmes par l'utilisation d'instructions vectorielles. Avant d'écrire un compilateur pour un tel langage, il faut explorer les possibilités des différents jeux d'instructions vectoriels du marché : AVX, NEON, et en particulier, RVV, une extension vectorielle de l'ISA RISC-V[1].

1.3. Objectifs du stage

Dans le contexte du projet **SxC**, ce stage consiste d'abord à explorer les outils disponibles permettant de travailler avec RVV, puis de réaliser des mesures exploratoires des performances de différents algorithmes implémentés en RVV. Les produits attendus à la fin de ces huit semaines sont :

1. La mise en place d'une chaîne d'outillage RISC-V permettant d'utiliser RVV
2. Des implémentations RVV de différents algorithmes de traitement de flux
3. Un moyen de mesurer les performances de programmes RVV

La section 2 donne une vue générale de la programmation vectorielle, de ses avantages et de ses inconvénients, et montre quelques exemples de programmes écrit avec RVV. Ensuite, la section 3 explore l'écosystème RISC-V actuel, en s'intéressant notamment aux différents outils disponibles pour la simulation fonctionnelle et la mesure de performance. Elle fait aussi état des difficultés rencontrées à chaque étape du cycle de développement d'applications utilisant RVV. La section 4 décrit le processus d'implémentation en RVV de quelques algorithmes de traitement de flux. Enfin, la section 5 examine les performances de ces différentes implémentations.

2. La programmation vectorielle

Dans un souci d'amélioration des performances, la tendance des quelques dernières décennies chez les concepteurs de micro-processeurs fut d'abord d'augmenter le nombre d'instructions exécutées par seconde, mais cette technique commence malheureusement à montrer ses limites. De plus en plus, ces concepteurs tentent d'encourager l'exploitation du parallélisme « local » de certains types de données, notamment à travers l'ajout d'instructions dites **SIMD** (*Single Instruction, Multiple Data*), qui permettent de traiter avec une seule instruction un bloc de données souvent quatre ou huit fois plus grand que ce qui est manipulable habituellement. Les programmes utilisant ce type d'instructions sont dits *vectoriels*, puisque les données sont traitées comme une liste de petits blocs unidimensionnels, similaires à des vecteurs de données.

[1] https://riscv.org/

2.1. Concept de base

Prenons par exemple une fonction **invertBits** permettant d'inverser une liste de bits. Traditionnellement, on écrirait un programme *scalaire* (parfois aussi appelé *séquentiel*) traitant chaque item l'un après l'autre de la façon suivante:

invertBits (bits: **bit[]**, bitCount):
1 **for** $i \leftarrow 0$ **to** bitCount $- 1$
2 bits[i] = **not**(bits[i])

Algorithme 1. – Pseudo-code scalaire pour **invertBits**

Cependant, la vision vectorielle viserait plutôt à traiter bits comme étant une suite de *vecteurs de bits*, sur lesquels l'opération d'inversion de bits **vectorNot** est déjà implémentée. On aurait alors un pseudo-code similaire, mais manipulant cette fois-ci des *vecteurs* d'éléments plutôt que des éléments singuliers.

invertBits (bitVecs: **Vector<bit>[]**, vectorCount):
1 **for** $i \leftarrow 0$ **to** vectorCount $- 1$
2 bitVecs[i] = **vectorNot**(bitVecs[i])

Algorithme 2. – Pseudo-code vectoriel pour **invertBits**

La fig. 1 montre la différence d'exécution entre l'algorithme 1 et l'algorithme 2 en faisant apparaître les « blocs » contenant plusieurs éléments à la fois.

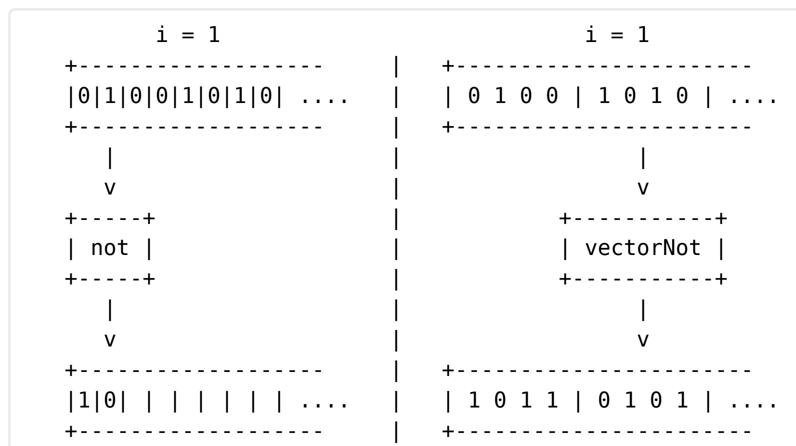

Fig. 1. – Représentation de la progression de l'algorithme 1 scalaire (à gauche) et de l'algorithme 2 vectoriel (à droite).

Remarquons qu'une des étapes absentes de l'algorithme vectoriel est la « conversion » entre **bit[]** et **Vector<bit>[]**. En effet, en pratique une implémentation vectorielle reçoit généralement un tableau de bits et doit s'occuper elle-même de synthétiser des vecteurs à partir de celui-ci. Il faut également qu'elle soit capable de propager ou stocker le résultat d'une opération vectorielle.

2.2. Programmation vectorielle sur x86

2.2.1. Le vecteur le plus basique : `int`

En réalité, il existe déjà un type **Vector<bit>** extrêmement répandu : `int`. Bien qu'il soit plus souvent vu simplement comme un nombre, il est aussi possible de le traiter comme un vecteur de bits : en effet, il existe en C quelques opérations opérant sur les bits individuels d'un `int`, comme `&` (AND), `|` (OR), `^` (XOR) ou `~` (NOT, qui est d'ailleurs l'équivalent de la fonction **vectorNot** de l'algorithme 2). Sous cet angle, un `int` est un vecteur de taille fixe de 32 bits, permettant d'opérer sur des bits.

Il n'est cependant pas possible de manipuler d'autres types/tailles de données : bien que 32 bits puissent techniquement contenir quatre `char`s de 8 bits chacun, il n'est généralement pas possible d'appliquer une opération sur chaque `char` indépendamment. Un `int` ne peut donc pas être utilisé pour, par exemple, additionner deux listes de `char`s, puisqu'il n'existe pas d'opération additionnant séparément les quatres `char`s pouvant être contenus dans un `int`.

2.2.2. Vecteurs SSE et AVX

Sur x86, les jeux d'instructions SIMD les plus répandus sont **SSE**[1] et **AVX** / **AVX2**[2,3], permettant de manipuler des vecteurs de 128 bits et 256 bits, respectivement. Ces extensions[4] offrent des opérations sur des vecteurs d'entiers de 8, 16, 32 et 64 bits, et sur des nombres à virgules flottantes de 32 et 64 bits.

Chaque vecteur correspond exactement à un registre vectoriel, et le type de données sur lequel une opération s'effectue est encodé dans l'instruction elle-même. L'instruction AVX2 `vpcmpeqb`, par exemple, permet d'effectuer une comparaison (`cmp`) d'égalité (`eq`) sur des entiers (`p`acked integer) 8-bits (`b`yte). Si l'on souhaitait à la place opérer sur des entiers de 32-bits, on utiliserait `vpcmpeqd`, avec le suffixe `d` pour *double-word*, i.e. 32-bits sur x86. Il est donc possible d'utiliser successivement deux opérations sur différents types de données, simplement en utilisant les instructions correspondantes.

Les vecteurs SSE et AVX ont une taille fixe, c'est-à-dire qu'il n'est pas possible de manipuler uniquement une partie du vecteur : toute opération se fait sur l'entièreté du vecteur. En particulier, il n'est pas possible de charger moins de 128 bits (resp. 256 bits) dans un vecteur. Ceci impacte la conversion entre tableau d'éléments et vecteur, puisqu'il faut maintenant également gérer le cas où le nombre d'élément n'est pas un multiple de la taille de vecteur. Autrement dit, si la taille du tableau est $T_{\text{tab}} = n \cdot T_{\text{vecteur}} + k$, il ne sera pas possible de traiter les k derniers éléments avec un vecteur. La solution habituelle est de traiter d'abord le plus d'éléments possibles avec des opérations vectorielles, puis d'appliquer une implémentation scalaire (séquentielle) pour les k derniers éléments. La section 4 présente quelques implémentations AVX d'algorithmes basiques, qui utilisent cette approche.

[1] https://fr.wikipedia.org/wiki/Streaming_SIMD_Extensions
[2] https://fr.wikipedia.org/wiki/Advanced_Vector_Extensions
[3] Sauf indication contraire, le reste de ce document utilise les termes AVX et AVX2 interchangeablement.
[4] SSE a connu plusieurs itérations ; ce n'est qu'à partir de la deuxième version, SSE2, qu'il est possible d'opérer sur des vecteurs d'entiers 8/16/32/64 bits ou de flottants 64 bits.

2.3. Programmation vectorielle sur RISC-V

L'extension vectorielle RISC-V, souvent abrégée **RVV**[1], est un jeu d'instructions SIMD. Une de ses caractéristiques notables est sa flexibilité. En particulier, la taille des vecteurs peut varier entre processeurs; de même pour les types d'éléments supportés. La spécification définit d'ailleurs plusieurs « variantes » de l'extension, certaines permettant de spécifier la taille minimale des vecteurs, d'autres étant conçues pour les processeurs embarqués et n'intégrant donc qu'une partie des fonctionnalités de l'extension entière. Sauf indication contraire, le reste de ce rapport utilise *RVV* pour désigner la variante dite « applicative » de l'extension, qui requiert des vecteurs de 128 bits minimum, pouvant contenir des entiers de 8, 16, 32 et 64 bits, ainsi que des flottants de 32 et 64 bits.

RVV diffère de SSE et AVX en de nombreux points, le premier étant la taille des registres vectoriels : là où SSE et AVX ont des registres de 128 bits (resp. 256 bits) dans tous les processeurs x86, RVV laisse au fabriquant le choix de la taille des registres vectoriels, tant qu'ils font moins de 2^16 bits (65536 bits). Puisque la taille des vecteurs peut être aussi grande et variée, RVV permet d'opérer sur des vecteurs partiels, à l'opposé de SSE/AVX. Par exemple, même sur une machine avec des registres vectoriels pouvant contenir 128 octets (1024 bits), il est possible d'opérer sur un vecteur d'uniquement 107 octets. RVV pousse cette dissociation registre/vecteur encore plus loin en introduisant le concept de *multiplicité* des vecteurs : en RVV, il est possible de « regrouper » plusieurs registres vectoriels et de les traiter comme un seul vecteur. Par exemple, un vecteur ayant une multiplicité de 2 signifie que celui-ci occupe en réalité 2 registres, qui sont alors comme « fusionner ». On réfère alors au premier registre du groupe pour désigner le vecteur entier, et les autres registres ne peuvent pas être accédés. La fig. 2 illustre ce cas avec des vecteurs d'éléments de 8 bits.

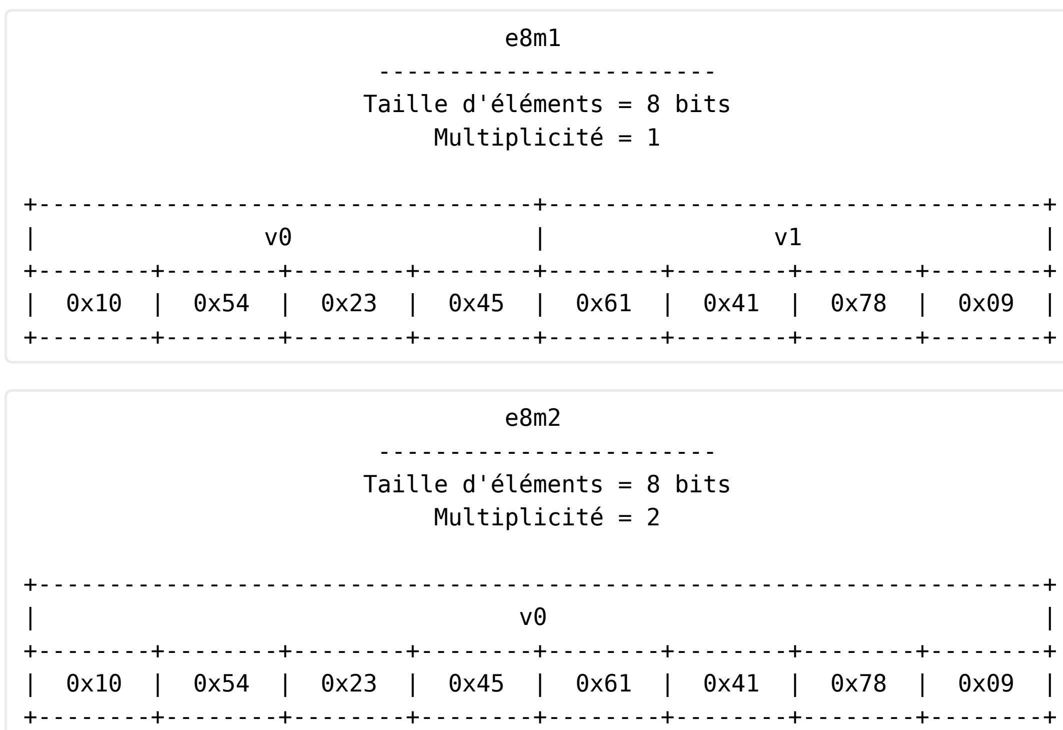

Fig. 2. – Illustration de l'effet de la multiplicité. En haut, deux vecteurs d'éléments de 8 bits, chacun avec une multiplicité de 1, contenus dans deux registres différents (`v0` et `v1`). En bas, un seul vecteur, avec une multiplicité de 2, « fusionnant » `v0` et `v1`.

[1] Le nom « RVV » désigne tout d'abord l'extension, mais, par abus de langage, on utilise aussi souvent « RVV » comme le nom d'un dialecte ou paradigme (par ex. « du code [écrit en] RVV » au lieu de dire « du code utilisant les instructions RVV »).

Étant donné que RVV supporte 7 multiplicités et 6 types d'éléments différents, il n'est pas envisageable de créer des instructions distinctes pour chaque combinaison opération-type-multiplicité. Ainsi, contrairement à SSE/AVX, RVV encode ces informations dans un état global, géré à l'aide de l'instruction dédiée `vsetvl`. En plus de s'occuper du type et de la multiplicité des vecteurs, celle-ci permet également de gérer la taille des vecteurs : on annonce le nombre total d'éléments que l'on souhaite traiter, et l'instruction nous renvoie le nombre maximum d'éléments pouvant être contenus dans un vecteur (en prenant compte du type et de la multiplicité). Étant donné la taille variable des registres vectoriels, cette information est cruciale, car elle permet de savoir combien d'items sont traités par chaque itération.

Une implémentation RVV annotée de l'algorithme 2 est présentée sur le listing 1. Cette implémentation est écrite en C pour faciliter la compréhension, en utilisant des fonctions pour modéliser les instructions correspondantes. (L'annexe A.1 contient une table de correspondance ainsi que quelques explications supplémentaires). Puisqu'il n'existe pas de type 1-bit en C, nous utilisons un tableau d'octets (`char`) à la place. On voit ici les mécanismes traités précédemment : à la ligne 8, on déclare le type des éléments (entiers de 8-bits) et la multiplicité du vecteur (`m8`), et on donne le nombre d'éléments totaux que l'on souhaite traiter. En retour, on obtient le nombre d'éléments qui seront réellement contenus dans un vecteur. Cette configuration s'appliquera à toutes les opérations suivantes, jusqu'à l'exécution de la prochaine instruction `vsetvl`. On charge ensuite les éléments dans un vecteur (ligne 11), en utilisant le nombre d'éléments traités jusqu'ici pour savoir où démarrer le chargement. On peut ensuite appliquer les opérations que l'on désire : ici, on inverse les bits des éléments du vecteurs avec **`vnot`**, puis on stocke les résultats avec **`vse8`**. Une fois que l'on a fini de traiter nos vecteurs, on incrémente le compteur d'éléments traités, puis, s'il reste des éléments non-traités, on réitère.

```c
1   void invertBits(char *src, size_t length) {
2     size_t itemsProcessed = 0;
3
4     while (itemsProcessed < length) {
5       // declare that we're trying to process `length - itemsProcessed`
6       // elements, each of size 8 bits; we also group vectors by 8,
7       // so that we can load as many elements as possible
8       size_t vl = vsetvl_e8m8(length - itemsProcessed);
9
10      // load 8-bits elements from `src`
11      vu8m8_t chunk = vle8(src + itemsProcessed);
12
13      // create a new vector with inverted bits
14      vu8m8_t invertedChunk = vnot(chunk);
15
16      // write the items back to the array
17      vse8(src + itemsProcessed, invertChunk);
18
19      itemsProcessed += vl;
20    }
21  }
```

Listing 1. – Implémentation RVV de l'algorithme 2. Les noms en gras sont des macros correspondantes à une instruction RVV chacune (cf annexe A.1).

3. L’environnement RISC-V et son outillage

Le jeu d’instructions RISC-V est principalement présent dans des contextes embarqués. Bien qu’il existe quelques machines « de bureau » basées sur un processeur RISC-V[1,2], celles-ci sont encore très loin d’être accessibles en terme de prix, et sont encore peu supportées par la grande majorité des applications et librairies. Ainsi, avant de pouvoir développer et tester des programmes RISC-V (et notamment RVV), nous devons mettre en place une **chaîne de développement** *cross-platform*. Une chaîne de développement, comme représentée dans la fig. 3, aussi appelé *toolchain* (lit. chaîne d’outillage), est un ensemble d’outils permettant l’écriture, la compilation, l’exécution, et le débuggage d’un programme. On dit que celle-ci est *cross-platform* lorsqu’elle est destinée à être installée sur une architecture différente de celle pour laquelle le programme est développé.

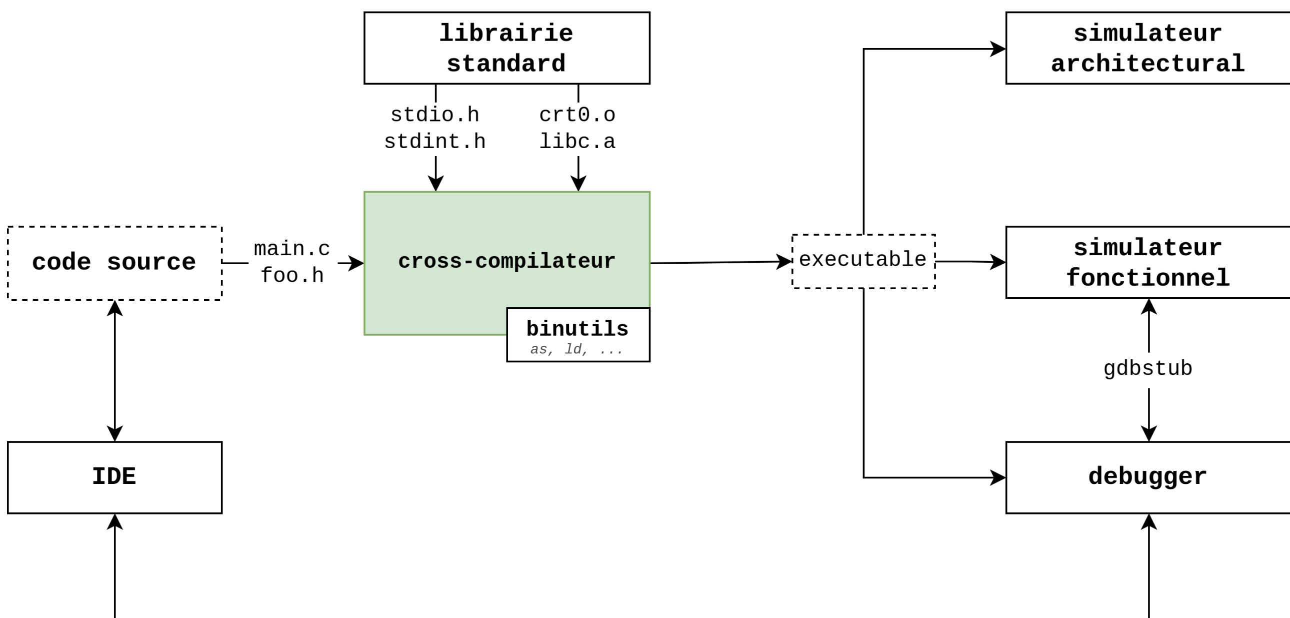

Fig. 3. – Les différentes composantes d’une chaîne de développement *cross-platform*

Dans la première partie du stage, j’ai mis en place une chaîne de développement similaire à la fig. 3 :

1. Cross-compilateur – Version de GCC compilée spécifiquement pour pouvoir générer des binaires statiques RISC-V (et particulièrement RVV)
2. binutils – Version de GNU Binutils compilée pour RISC-V
3. Librairie standard – Librairie standard C `newlib`[3] pour systèmes POSIX embarqués
4. Simulation fonctionnelle – Simulateurs `spike`[4] (avec le noyau `pk`) et `qemu-riscv` permettant d’émuler un système POSIX exécutant un programme RVV
5. Simulation architecturale – Simulation de design de processeurs supportant RVV, permettant d’obtenir une approximation plus réaliste du nombre de cycles d’un programme RVV; dans notre cas, ce fut le coprocesseur Vicuna [PP21] intégré avec le processeur CV32e40x[5]
6. Debugger – Version de GDB compilée pour RISC-V

[1] https://milkv.io/pioneer
[2] https://milkv.io/jupiter
[3] https://sourceware.org/newlib/
[4] https://github.com/riscv-software-src/riscv-isa-sim
[5] https://github.com/openhwgroup/cv32e40x

Le reste de cette section est dédiée aux rôles des différents outils ainsi qu'à la justification des choix effectués.

3.1. Exécution de programmes RVV

De par la nouveauté de l'extension RVV, les quelques systèmes la supportant sont particulièrement chers et difficiles à obtenir; nous avons donc préférer nous appuyer uniquement sur des outils de simulation. Nous distinguons cependant deux types distincts de simulation : la simulation *fonctionnelle* et la simulation *architecturale* (aussi appelée émulation). La simulation *fonctionnelle* permet de tester la correction d'un programme, indépendamment de sa cible d'exécution finale, et privilégie la vitesse d'exécution, quitte à sacrifier certains détails non-fonctionnels (par ex. les délais d'exécution de certains instructions ou la latence mémoire). Nous avons choisi pour celle-ci les outils `spike` et `qemu`, qui sont standards dans le domaine. La simulation *architecturale*, quant à elle, vise à combler les manques de la simulation fonctionnelle, en émulant du mieux possible ces « détails non-fonctionnels » (dont notamment les différents délais et particularités d'un système réel). Ce type de simulation peut-être particulièrement utile lorsqu'il y a nécessité de modéliser également l'interaction entre différents composants ou systèmes, pour laquelle la durée de chaque opération est critique.

Comme mentionné dans la section 1.3, un des objectifs de ce stage est la mesure de performance de courts programmes RVV. Cela requiert donc d'effectuer une simulation architecturale, émulant de manière réaliste l'interaction entre les différents composants d'un système réel. Celle-ci nous permettrait alors d'avoir un ordre d'idée des gains de performance, **si tant est qu'on puisse trouver une simulation suffisamment fidèle**. Nous avons cherché des designs *open-source* de processeurs supportant RVV 1.0. Contrairement à des architectures comme x86 ou ARM, l'écosystème RISC-V est particulièrement ouvert, et il existe donc pléthore de designs, chacun plus ou moins sophistiqués, performants, ou de niche. Ces recherches ont débouché sur deux designs, l'un surnommé Ara2 et l'autre Vicuna.

Ara2 [PER+22] est un coprocesseur vectoriel (c-à-d s'attachant à un coeur RISC-V classique, ici le CVA6) implémentant la version 1.0 « applicative » de l'extension RVV (en opposition à sa variante « embarquée »), et dont le design est clairement orienté vers la performance énergétique et temporelle. Son dépôt GitHub est également très bien documenté et organisé. Malheureusement, nous ne fumes pas capables de compiler ce design à cause d'un bug interne au compilateur. Après de nombreuses heures à tenter en vain de le résoudre ou de le contourner, nous avons mis ce design de côté et avons cherché des alternatives.

Vicuna [PP21] est, comme Ara2, un coprocesseur vectoriel. Contraitement à Ara2, il vise plutôt un contexte embarqué, et n'implémente donc pas l'entière de la variante « applicative » de RVV. À la place, seule la variante `Zve32x` est implémentée, ne supportant pas les opérations sur les flottants ou sur les entiers de 64 bits. Dans notre cas, cela ne sera pas un problème puisque les programmes que nous comptons exécuter traitent principalement des caractères (de 8 bits). Comme Ara2, il est distribué sur GitHub sous forme de fichiers source Verilog facilement simulables. L'aspect « minimaliste » de Vicuna en fait un design relativement compréhensible, malgré le fait qu'il ne soit pas autant documenté qu'Ara2. Ceci fut particulièrement pratique lorsque nous avons rencontré des bugs d'exécution, puisqu'il était alors possible d'aller regarder le code source pour savoir si le problème se trouvait dans Vicuna ou ailleurs. Un autre avantage de ce minimalisme est la rapidité de

simulation du design, ce qui était bienvenu lorsque fut venue l'heure des mesures de performances et de l'écriture de variantes (cf section 5).

Combiné au CV32e40x, Vicuna émule un système constitué uniquement d'un processeur, d'une mémoire vive, et d'un port UART, rien de plus. Ainsi, par défaut, nous n'avons ni système d'exploitation ni librairie standard, ce qui nous force à écrire certaines fonctions nécessaire à la compilation et l'exécution du programme. Par exemple, quelques soit les arguments passés, GCC *requiert* la présence des fonctions `memcpy`, `memcmp`, `memset` et `memmove`, même si elles ne sont pas explicitement utilisées. Pour interfacer avec le monde extérieur, nous ne pouvons pas juste nous reposer sur les fonctions « magiques » qu'offrent `spike` et `qemu`, et nous avons donc besoin d'implémenter notre propre version (minimale) de `puts`/`printf` utilisant le port UART simulé pour communiquer.

Avoir accès à des simulateurs fonctionnels **et** architecturaux permet d'accomplir des objectifs complémentaires:

1. `spike` et `qemu` permettent d'avoir une boucle de développement plutôt rapide, pour vérifier la correction du code RVV; `qemu` était particulièrement utile pour débugger grâce à son intégration avec les outils standard (par ex. GDB). Remarquons ici que le simulateur `spike`, qui est censé implémenter la spécification exacte, ne la respecte pas à certains endroits ; par ex. il est impossible de configurer des vecteurs de taille > 4096 bits, alors que la spécification autorise jusqu'à 2^{16} bits, ce qui rend plus difficile l'exploration complète du jeu d'instructions.
2. Vicuna nous permet d'avoir des mesures de performances plus réalistes, mais aussi de voir l'impact des différentes caractéristiques architecturales (taille des vecteurs, latence mémoire, caractéristiques du pipeline…) sur ces performances. Les variations que l'on peut explorer avec Vicuna sont beaucoup plus nombreuses que pour les simulateurs fonctionnels, qui eux n'exposent que quelques paramètres (uniquement les tailles des vecteurs et des éléments).

Malheureusement, nous n'avons au final pas pu explorer les différentes configurations de Vicuna autant que prévu, à cause de multiples bugs dans l'architecture du coprocesseur ainsi que dans le simulateur utilisé.

3.2. Outillage additionnel et tâches accomplies

Dans le contexte de ce projet, la mise en place d'une chaîne d'outillage ne s'arrête pas simplement au téléchargement et à la compilation des outils. Il faut ensuite les coordonner correctement et offrir aux utilisateurs un moyen simple d'interfacer avec ceux-ci. Dans mon cas, ceci incluait :

1. L'automatisation de l'installation d'une chaîne de cross-compilation RVV (comportant les éléments précédents)
   - Cela comprend le clonage du code source des différents outils, leur configuration, leur compilation et leur installation harmonieuse
2. Un système de compilation d'un programme pour différentes cibles : RISC-V non-vectoriel, RVV, x86 non-vectoriel, AVX2
   - Ce système est constitué de fichiers sources partageant le code commun ainsi que les déclarations des méthodes algorithmes, faisant office d'interface. On implémente ensuite cette interface pour chaque plateforme (par exemple, un fichier `avx.c` qui contient des fonctions écrites avec des intrinsèques AVX2). Le tout est orchestré par un simple makefile utilisant la bonne *toolchain* et les bonnes options de compilation pour chaque cible.

3. L'automatisation du lancement de programmes RVV sur différents simulateurs, avec opportunité de debug rapide, interfaçant optionnellement avec `vscode`
   - Règles supplémentaires dans le makefile central
   - Template de configuration `vscode` pour le debug
4. L'automatisation de la visualisation des performances des benchmarks
   - Scripts utilisant la bibliothèque `matplotlib` pour tracer des histogrammes comparant les différentes implémentations d'un même algorithme
5. La modification de différents outils pour faciliter leurs utilisations
   1. Patch de `spike` pour permettre de tester avec des tailles de vecteurs plus grandes
   2. Réécriture des makefiles de Vicuna pour améliorer l'expérience utilisateur
   3. Écriture de routines standards pour faciliter l'exécution des programmes sur Vicuna (cf section 3.1)

Ces automatisations sont cruciales : une grande partie de ces outils requièrent une longue durée de compilation, qu'il faudra relancer à chaque changement de configuration. Pour GCC, ajouter ou retirer une partie du *backend* nous oblige à le recompiler, ce qui peut prendre plusieurs dizaines de minutes à chaque fois, utilisant l'entièreté des ressources de la machine de compilation, la rendant inutilisable pour d'autres tâches; LLVM, un autre compilateur compatible avec RVV, est encore plus long à compiler, réservant souvent la machine pendant plusieurs heures. Il n'est aussi pas toujours simple de savoir quelles fonctionnalités sont compatibles avec d'autres (par exemple, certaines extensions ne sont pas disponibles pour les processeurs RISC-V 32-bits).

4. Réécriture d'algorithmes vers l'extension vectorielle RISC-V

Une de mes missions lors de ce stage était d'implémenter des algorithmes simples en utilisant RVV. Pour m'aider, chaque algorithme était décrit à la fois en langage naturel ainsi qu'en pseudo-code ayant une approche plutôt « traitement de flux ». Une partie d'entre eux étaient des fonctions de la libraire standard C (`memchr`, `memmem`, `memcmp`), tandis que d'autres étaient des algorithmes utiles dans le traitement de texte ou le parsing (`memseq`, `dyck`, …). Dans cette section, nous explorons brièvement ce qui motive cette tâche, puis montrons quelques exemples de réécriture pour illustrer notre propos.

4.1. Dans quel but ?

Étant donné que ce projet a pour but la création d'un langage facilitant l'écriture de code vectoriel, il est essentiel d'étudier d'abord les caractéristiques des différentes extensions vectorielles. La conception d'un optimiseur et d'un générateur de code visant plusieurs plateformes nécessite de savoir à l'avance les concepts que l'on peut « facilement » abstraire et traduire pour chaque cible et ceux qui demanderont plus de finesse ou de spécialisation. Alors que mon collègue de stage à VERIMAG, Adnane El-Asli, avait pour mission d'explorer l'expression d'opérations vectorielles dans l'infrastructure LLVM, mon stage était en partie dédié à relever les différences entre RVV et les extensions vectorielles x86 (SSE et AVX).

4.2. Un exemple pratique : `memchr`

Un algorithme utilisé presque universellement est la recherche d'éléments dans un tableau. Ces fonctions prennent généralement en entrées un tableau, sa taille, et l'élément à trouver, et retourne la position de ce dernier, ou une valeur spéciale s'il n'est pas trouvé (par ex. -1).

La fonction privilégiée pour ce genre de tâches, spécifiée par le standard C, est `memchr`. Elle retourne un *pointeur* vers la première occurrence (plutôt qu'une position relative au tableau), et retourne `NULL` si le caractère n'est pas présent.

memchr (str, N, c):
1 **for** $i \leftarrow 0$ **in** $N - 1$ **:**
2 **if** str$[i] = c$ **:**
3 **return** i
4 **return** -1

Algorithme 3. – Pseudo-code pour `memchr`

Une implémentation scalaire (non-vectorielle) de l'algorithme 3 est présentée dans le listing 2.

```
1  char* memchr(char *str, size_t n, char c) {
2    for (size_t i = 0; i < n; i++) {
3      if (str[i] == c)
4        return &(str[i]);
5    }
6
7    return NULL;
8  }
```

Listing 2. – Implémentation scalaire de `memchr`

4.2.1. Approche vectorielle et implémentation AVX

Étant donné sa prévalence et sa simplicité apparente, c'est un des premiers algorithmes que j'ai tenté d'écrire en RVV. Pour m'aider à démarrer, on m'a fourni une implémentation vectorielle utilisant le jeu d'instructions AVX2 (listing 3).

```c
char* memchr(char *str, size_t n, char c) {
  vector c_mask = load_mask(c);

  for (size_t i = 0; i + vector_size <= n; i += vector_size) {
    vector chunk = uload(str + i); // load the chunk of the stream
    vector c_in_chunk = vectorEq(chunk, c_mask); // compare bytewise
    mask offsets = maskFromVector(c_in_chunk); // compute the bitmask
    if (offsets != 0) {
      // ctz returns the offset of the left most 1 in mask
      int first_offset = ctz(offsets);
      return &(str[i + first_offset]);
    }
  }

  // a sequential variant for the end of the block
  size_t trailingIdx = n - (n % vector_size);
  for (size_t i = trailingIdx; i < n; i++)
    if (str[i] == c) return &(str[i]);
  return NULL;
}
```

Listing 3. – Implémentation en AVX2 de `memchr`. Les noms en gras sont des macros correspondantes à une instruction AVX2 chacune (cf annexe A.2).

Le flux d'entrée est découpé en multiples blocs (ici appelés `chunk`s) de taille **`vector_size`** (constante égale à 32, nombre d'octets contenus dans un vecteur en AVX2). Ces blocs sont chargés grâce à **`uload`**. Ensuite, la fonction **`vectorEq`** permet de connaître les positions d'égalités de deux vecteurs, en créant un nouveau vecteur contenant des 1 (entier 8-bits) à ces positions, et des 0 ailleurs. Dans notre cas, on compare le chunk avec un vecteur constant à `c` (initialisé à la ligne 2). On « compresse » ensuite le vecteur résultant de cette comparaison dans un registre général (de 32 bits), en faisant correspondre un 1 ou 0 entier par la valeur booléenne correspondante dans chaque bit du registre (ici appelé `offsets`). Pour savoir s'il y a une occurrence de `c`, on a simplement à vérifier si `offset` est différent de zéro, et si c'est le cas, on peut récupérer la position du premier `1` (ergo du premier `c`) à l'aide de `ctz` (ligne 10). La valeur de retour est donc égale à l'offset du bloc dans le flux original (`i`) ajouté à la position dans le bloc (`first_offset`).

La fig. 4 montre un exemple d'exécution de l'implémentation AVX de `memchr` pour rechercher le caractère `'s'` dans la chaîne `"hello john, how are you? how's the job? how are the kids?"`. À noter que la valeur de `offsets` est écrite de gauche à droite (bit 0 à gauche, bit 31 à droite), pour mieux s'accorder avec l'écriture des vecteurs, qui se fait plus naturellement de gauche à droite (tout comme la chaîne d'entrée) ; cependant, en réalité `offsets` est un entier de 32 bits.

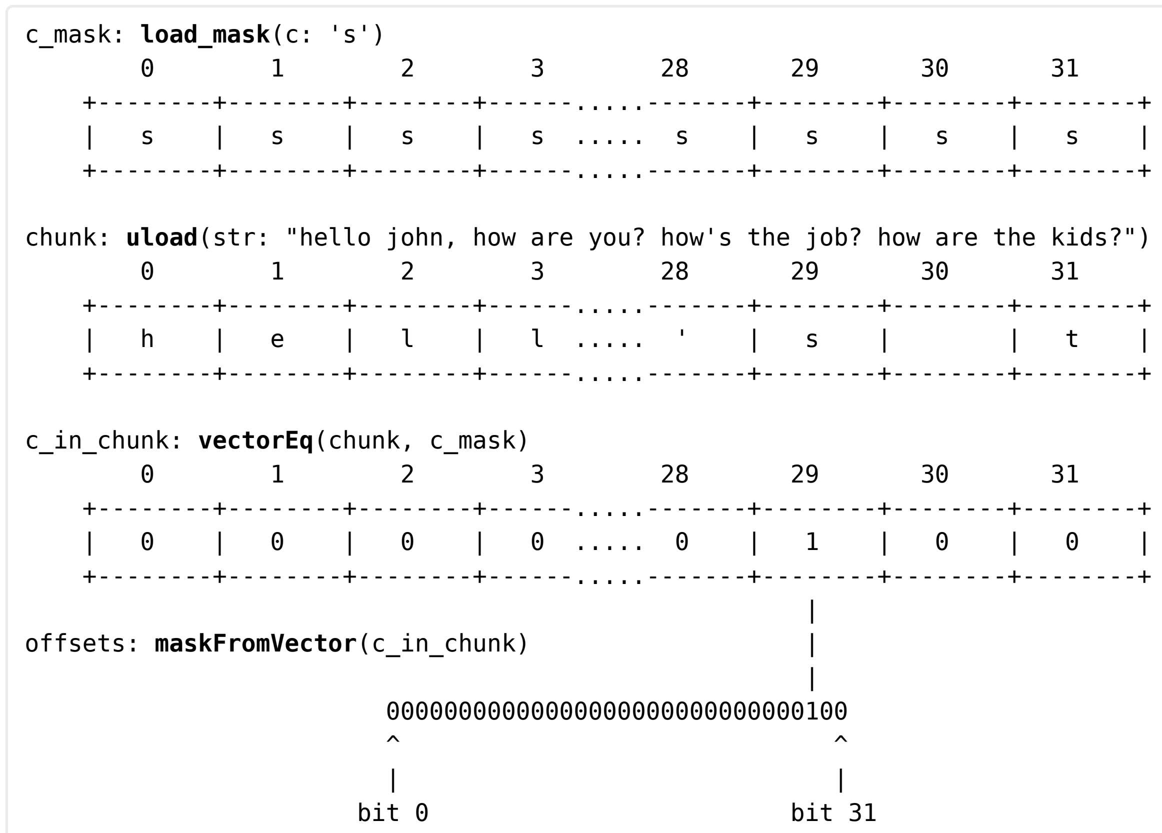

Fig. 4. – Représentation de l'exécution de l'implémentation AVX de `memchr` sur des données d'exemples

4.2.2. Implémentation en RVV

Pour la réécriture en RVV, la plus grosse différence est l'unification des boucles séparées « vectoriel vs. séquentiel » en une seule boucle de traitement de vecteurs de tailles variables (voir section 2.2.2 et section 2.3). Nous suivons le même schéma que dans l'exemple présenté dans la section 2.3, en utilisant des vecteurs de multiplicité 8 contenant des entiers 8-bits. Ensuite, comme dans la version AVX, on charge un bloc d'éléments (ligne 8), puis on les compare à `c`. Cette fois-ci, l'instruction `vmseq.vx` (ligne 10) permet de comparer directement un vecteur et un registre général, nous évitant ainsi de créer un vecteur additionnel comme avec **`vectorEq`** en AVX. On récupère alors un vecteur de bits (masque) `matches_mask`, qui n'a ici pas besoin d'être « compressé » comme précédemment (il est sous la même forme que `offsets` dans l'implémentation AVX). Enfin, l'instruction **`vfirst`** (ligne 12) permet de récupérer la position du premier `1` dans le masque, ou −1 s'il n'y en a aucun (dans l'example de la fig. 4, ce serait 29, puisque le 29 bit est le premier bit à 1).

```c
char* memchr(char *str, size_t n, char c) {
  size_t dataLeft = n;
  char *currStart = str;

  while (dataLeft > 0) {
    size_t vl = vsetvl_e8m8(dataLeft);
    // load `vl` 8-bits elements from `currStart`
    vu8m8_t chunk = vle8(currStart);
    // compare items in chunks with 'c' and put results in bitmask
    vbool1_t matches_mask = vmseq_vx(chunk, c);
    // first index of first bit set in mask
    long match_idx = vfirst(matches_mask);

    if (match_idx >= 0)
        return &(currStart[match_idx]);

    currStart += vl;
    dataLeft -= vl;
  }

  return NULL;
}
```

Listing 4. – Implémentation en RVV de `memchr`. Les noms en gras sont des macros correspondantes à une instruction RVV chacune (cf annexe A.1).

Comme le démontre cet exemple, le portage d'une implémentation AVX vers RVV est souvent relativement direct (une fois le jeu d'instructions RVV maîtrisé). Cependant, on voit tout de même que RVV simplifie une partie du code en enlevant la redondance de la boucle séquentielle ainsi que les échanges entre les registres vectoriels et généraux.

4.3. Un cas de réécriture plus difficile

Étant donné les différences entre AVX et RVV, tous les portages ne peuvent pas se faire aussi simplement. Un des atouts majeurs de RVV, sa taille de vecteurs variable, a parfois engendré des difficultés lors de la réécriture d'algorithmes prenant avantage d'une taille fixe relativement basse. Un domaine particulièrement affecté par ce changement est la gestion et le traitement des « masques », comme dans le cas de `memseq`.

`memseq` est un des algorithmes pour lesquels une implémentation AVX m'a été fournie (listing 5). Il s'agit de trouver dans une séquence la première instance de deux éléments consécutifs `a` et `b`, ou de retourner une valeur spéciale si `ab` n'existe pas dans l'entrée. C'est une extension assez naturelle de `memchr`, et son pseudo-code (algorithme 4) est extrêmement similaire.

memseq (str, N, a, b):
1 **for** $i \leftarrow 0$ **in** $N - 2$ **:**
2   **if** $\mathrm{str}[i] = a$ **and** $\mathrm{str}[i + 1] = b$ **:**
3   **return** i
4 **return** -1

Algorithme 4. – Pseudo-code pour `memseq`

L'implémentation vectorielle de cet algorithme n'est cependant pas aussi simple. Le traitement de données par blocs requiert de « ruser » quelque peu pour pouvoir effectuer l'équivalent de la ligne 2 du pseudo-code : d'un certain point de vue, cette comparaison revient à s'assurer que `b` est situé à la même place que `a`, mais dans une séquence « décalée » d'un élément. Comme nous le verrons par la suite, la version AVX préfère plutôt décaler les résultats que les sources, mais cela est à peu près analogue. Un autre problème émergeant du traitement en bloc est la gestion des occurrences de fin de bloc : si il y a un `a` en fin de bloc, nous devons examiner le prochain bloc pour savoir si c'est un match entier (c-à-d le premier élément du bloc est un `b`) ou non. Ceci ajoute encore de la complexité à notre implémentation, et interfère également avec le « décalage » discuté précédemment. Avec ce contexte posé, disséquons rapidement l'implémentation AVX avant de passer à la réécriture en RVV.

Le listing 5 présente l'implémentation AVX que l'on m'a fournie. De loin, celle-ci ressemble simplement à une implémentation entremêlées de `memchr` recherchant deux caractères dans une même séquence : on initialise deux vecteurs constants d'éléments à comparer (ligne 2 et ligne 3), puis on itère en chargeant des blocs l'on comparent ensuite avec chacun de nos vecteurs constants (ligne 8 et ligne 9), pour ensuite en faire des masques. La différence majeure est la ligne 14, qui remplit plusieurs fonctions en même temps :

1. l'opération `last_block_a_match | (offsets_a<<1)` décale les occurrences de `a` à gauche et affecte le bit de poids faible à 1 si `last_block_a_match` est vraie
   - cette variable est mise à jour à la ligne 19, et est vraie si le bloc précédent contenait un `a` en dernière position
2. l'opération `(...) & offsets_b` permet d'obtenir des `1` là où les occurrences décalées de `a` s'alignent avec les occurrences de `b`

En d'autre termes, `result` contient un masque indiquant les positions des `b` qui sont précédés par un `a`, même si ce `a` est dans le bloc précédent. Comme dans `memchr`, on regarde ensuite s'il y a une telle occurrence (ligne 16) puis sa position (avec `ctz`, ligne 17), bien qu'ici on décrémente la position par 1 car on veut renvoyer la position du `a` commençant la séquence, et non pas du `b`. S'il n'y avait pas d'occurrence de `ab` dans ce bloc, alors on affecte la valeur de `last_block_a_match` en décalant les occurrences de `a` dans le bloc actuel jusqu'à obtenir le bit de poids forts original en position du poids faible de `last_block_a_match` (ligne 19). Le reste de la fonction est une variante séquentielle, ici omise, permettant de traiter le dernier bloc de données comme était nécessaire pour `memchr`. L'annexe A.4.1 contient un exemple d'exécution de cet implémentation.

```c
char *memseq(const char *str, size_t length, char a, char b) {
  vector mask_a = load_mask(a);
  vector mask_b = load_mask(b);

  bool last_block_a_match = 0;
  for (size_t i = 0; i + vector_size < length; i += vector_size) {
    vector chunk = uload(str + i);
    vector a_in_chunk = vectorEq(chunk, mask_a);
    vector b_in_chunk = vectorEq(chunk, mask_b);

    mask offsets_a = maskFromVector(a_in_chunk);
    mask offsets_b = maskFromVector(b_in_chunk);

    mask result = ((last_block_a_match | (offsets_a<<1)) & offsets_b);

    if (result != 0)
      return &(str[i + ctz(result) - 1]);

    last_block_a_match = offsets_a >> (vector_size-1);
  }

  /* épilogue séquentiel omis pour brièveté */

  return NULL;
}
```

Listing 5. – Implémentation AVX2 de `memseq`

Comme on peut le remarquer, cette implémentation repose en partie sur le fait que les résultats de comparaisons soit facilement accessibles sous formes de GPR[1] classiques de 32 bits. Cela nous permet d'effectuer des décalages et des opérations logiques, extrêmement légères en terme de performance, et de n'utiliser uniquement les unités vectorielles (plus coûteuses) que lorsque nécessaire (c-à-d quand on compare 32x8 bits en même temps). Malheureusement, les vecteurs de tailles variables de RVV ne nous permettent pas de nous assurer que les masques résultants de comparaisons vectorielles pourront être stockés dans des GPR (ils sont potentiellement trop gros). De plus, bien qu'il existe quelques instructions pouvant effectuer des opérations bit-à-bit sur des masques, il n'y a aucune façon d'appliquer un décalage à un masque ; il n'est pas non plus possible de récupérer le dernier bit d'un masque ni de l'insérer dans un autre masque. Tous ces problèmes nous empêchent donc de traduire l'implémentation AVX directement.

Heureusement, RVV nous fournis les instructions `vslideup` / `vslidedown`, qui permettent de décaler les éléments à l'intérieur d'un vecteur par un offset donné. Par exemple, appliquer `vslideup` avec un décalage de 3 à un vecteur `v` déplace `v[0]` en `v[3]`, `v[1]` en `v[4]`, etc. (Les valeurs de `v[0]`, `v[1]` et `v[2]` restent alors inchangées.). On peut donc substituer le décalage du masque par un décalage des éléments avant la comparaison, bien que cela risque d'avoir un coût plus important en terme de performance. Une variante de ces instructions, `vslide1up` / `vslide1down`, permettent de décaler un vecteur d'un élément **et** d'insérer une valeur spéciale à la place de l'élément « manquant ». On peut donc l'utiliser pour insérer le dernier caractère du bloc précédent dans le vecteur qui est comparé avec `a`, ce qui résout notre deuxième problème de traduction !

[1]General Purpose Register – un type de registre générique, utilisé pour la plupart des opérations classiques

Le code RVV final est présenté dans le listing 6. Celui-ci utilise une variable permettant de se souvenir du dernier caractère du bloc précédent, qui est initialisée à une valeur différente de `a` (ici NON a, ligne 5). On charge ensuite un bloc à partir du flux d'entrée (ligne 8), puis on crée une version « décalée » du bloc (ligne 13), en insérant le caractère précédent `last_char` au début, comme discuté dans le paragraphe précédent. On cherche ensuite les occurrences de `a` et de `b`, en utilisant le chunk décalé pour `a` (ligne 14) et le chunk non-décalé pour `b` (ligne 15). Puis on procède comme en AVX : on applique un ET bit-à-bit entre les masques résultants (ligne 17), puis on recherche le premier `1` dans le nouveau masque (ligne 19), indiquant la position d'un `b` précédé d'un `a`. Si jamais il n'y avait aucun match, on passe alors à la prochaine itération, en faisant attention à stocker le dernier caractère du bloc (ou, de façon équivalent, le caractère avant le prochain bloc).

```c
1   char* memseq(const char *str, size_t strLength, char a, char b) {
2     size_t dataLeft = strLength;
3     const char *currStr = str;
4     // start with a value other than 'a' to avoid false positives
5     char last_char = ~a;
6     while (dataLeft+1 >= 2) { // +1 because we take `lastChar` into account
7       size_t vl = vsetvl_e8m8(dataLeft);
8       vu8m8_t chunk = vle8(currStr, vl);
9
10      // for the 'a' matches, we slide the chunk up by one, appending
11      // the last character from the last block (which we discarded
12      // with the slide of the last block/iteration)
13      vu8m8_t offset_chunk = vslide1up_vx(chunk, last_char);
14      vbool1_t a_matches = vmseq_vx(offset_chunk, a);
15      vbool1_t b_matches = vmseq_vx(chunk, b);
16
17      vbool1_t full_matches = vmand(a_matches, b_matches);
18
19      long match_idx = vfirst(full_matches, vl);
20      if (match_idx >= 0) {
21        // match idx points to 'b', so to point to 'a' we go back by one
22        return (char*)(currStr + (match_idx-1));
23      }
24
25      currStr += vl;
26      last_char = currStr[-1];
27      dataLeft -= vl;
28    }
29
30    return NULL;
31  }
```

Listing 6. – Implémentation RVV de `memseq`

4.4. Variantes et sélection d'instruction sous RVV

Avec le peu de temps restant, j'ai aussi essayé d'écrire différentes « variantes » de chaque algorithme. Par exemple, sous RVV, de nombreuses instructions comparants deux vecteurs peuvent être substituées par des comparaisons entre un vecteur et un GPR (ou même dans certains cas entre un vecteur et une valeur immédiate). Assez tôt lors des réécritures, je me suis donc demandé laquelle de ces formes était la plus performante. Bien qu'a ce moment-là, je n'avais pas encore réalisé l'infrastructure nécessaire à ce genre de comparaisons, j'ai écrit différentes variantes de chaque implé-

mentation. Pour reprendre l'exemple de `memseq`, la comparaison `vmseq` entre le bloc et les valeurs « constantes » (`a` et `b`) peut-être effectuée *vecteur-à-registre* (généralement dénoté par le suffixe `_vx`) ou *vecteur-à-vecteur* (dénoté par `_vv`). La comparaison `vv` requiert, sans surprise, deux vecteurs de plus : l'un contenant uniquement des `a` et l'autre uniquement des `b`. Ceci est similaire à la technique utilisée avec AVX pour les comparaisons, comme par exemple dans `memchr` (listing 3) à la ligne 2. En RVV, nous pouvons utiliser l'instruction `vmv.v.x` pour initialiser un vecteur avec une valeur venant d'un GPR. Une implémentation alternative utilisant cette technique est proposée par le listing 7. Celle-ci utilise `vmv_v_x` pour initialiser deux vecteurs contenant `a` (ligne 6) et `b` (ligne 7). Ils sont ensuite utilisés ligne 12 et 13 pour comparer les éléments du chunk.

```c
1  char* memseq(const char *str, size_t strLength, char a, char b) {
2    size_t dataLeft = strLength;
3    const char *currStr = str;
4    char last_char = ~a;
5    size_t max_vl = vsetvl_e8m8(dataLeft);
6    vu8m8_t a_vec = vmv_v_x(a, max_vl);
7    vu8m8_t b_vec = vmv_v_x(b, max_vl);
8    while (dataLeft+1 >= 2) {
9      size_t vl = vsetvl_e8m8(dataLeft);
10     vu8m8_t chunk = vle8(currStr, vl);
11
12     vbool1_t a_matches = vmseq_vv(vslide1up_vx(chunk, last_char), vec_a);
13     vbool1_t b_matches = vmseq_vv(chunk, vec_b);
14
15     vbool1_t full_matches = vmand(a_matches, b_matches, vl);
16
17     long match_idx = vfirst(full_matches, vl);
18     if (match_idx >= 0) {
19       // match idx points to 'b', so to point to 'a' we go back by one
20       return (char*)(currStr + (match_idx-1));
21     }
22
23     currStr += vl;
24     last_char = currStr[-1];
25     dataLeft -= vl;
26   }
27
28   return NULL;
29 }
```

Listing 7. – Implémentation RVV de `memseq` utilisant la comparaison *vecteur-à-vecteur* plutôt que *vecteur-à-registre*. En vert, les changements effectués par rapport à l'implémentation précédente.

Selon le modèle que l'on se fait du processeur vectoriel, cette approche peut paraître intuitivement plus ou moins performante comparée aux opérations *vecteur-à-registre* : on peut s'imaginer qu'en interne, lors d'une opération `vx`, le processeur charge d'abord un vecteur avec la valeur du registre *puis* fait la comparaison ; dans ce cas il parait alors naturel de considérer les opérations `vv` plus performantes sur le long terme. D'un autre point de vue, cependant, il est possible que le processeur puisse accélérer la comparaison lorsqu'il n'a besoin de manier qu'un seul vecteur à la fois plutôt que deux, car ceci réduirait alors le nombre d'accès au banc de registres vectoriels.

Quelle que soit l'architecture du processeur, l'utilisation de vecteurs additionnels risque une raréfaction des registres (ce phénomène est généralement appelé"pression registre"). Lorsqu'il n'y a pas assez de registres pour contenir toutes les variables manipulées, on se trouve obligés de les stocker et charger en mémoire pendant nos calculs (un phénomène appelé *register spilling*, lit. débordement de registres), ce qui peut être très coûteux. Pour éviter cela, on peut réduire la multiplicité de nos vecteurs, et donc le nombre de registres occupés par chaque vecteur, mais ceci divise le nombre d'éléments traités par itération, et donc le « rendement ».

Dans le cas de `memseq`, il se trouve qu'avec une multiplicité de `m8`, l'ajout de deux nouveaux vecteurs pour les comparaisons applique une trop grande pression, et le compilateur est forcé de causer un *spill* des registres `a_vec` et `b_vec`[1]. Ainsi, sans avoir besoin de mesurer, il est raisonnable de prédire que cette version de `memseq` aura une performance moindre. Nous reviendrons sur la validation de telles prédictions à la section 5.

4.5. Résultats de cette réécriture

Malgré les quelques problèmes illustrés dans la section précédente, la plupart des réécritures étaient relativement directes. Pour les algorithmes `mask`, `memcmp` et `memmem`, le code (annoté) est donné en annexe A.3. Ce processus de réécriture m'a permis de bien me familiariser avec l'extension RVV, autant dans les détails de ses instructions que dans ses concepts plus abstraits. Ceci m'a donné une certaine appréciation pour la façon dont RVV est conçue, qui est plutôt élégante, bien qu'il y a clairement quelques instructions à mon avis manquantes, comme par exemple plus d'opérations bit-à-bit sur les masques, ou la possibilité de récupérer un élément à un certain index d'un vecteur. RVV offre également un plus grand espace d'exploration en terme de choix d'instruction (comme démontré dans la section 4.4), ce qui a ses avantages et inconvénients lors de l'optimisation de code : d'un côté, des instructions plus spécifiques et qui couvrent plus fonctionnalités offrent plus d'opportunités d'optimisation au compilateur, mais d'un autre côté cela complexifie la sélection et l'ordonnancement des instructions en augmentant le nombre les possibilités à explorer. La multiplicité des vecteurs est aussi un paramètre de plus que nous pensons avoir une grande influence sur les optimisations les plus tardives, telles que l'allocation de registres ou la sélection d'instructions. Bien qu'il y ait déjà quelques avancées à ce sujet (par ex. [SHI+23]), cela restera une zone sensible lors de la conception d'un *backend* RVV.

5. Mesures et performances

Après s'être assuré de la correction *fonctionnelle* de nos implémentations, nous désirons désormais évaluer leurs performances.

Les mesures de performances de code RVV sont extrêmement rares, étant donné le manque d'implémentations physiques accessibles. Les quelques relevés disponibles sont souvent limités uniquement à des algorithmes algébriques, et non pas de traitement de flux comme nous le souhaitons. Il existe bien quelques mesures d'algorithmes plus proches des nôtres, comme par exemple la conversion de texte UTF-8 en UTF-16[2], mais celles-ci sont généralement faites sur d'anciennes versions de l'extension. Ces mesures sont loin d'être simples, notamment sur du matériel physique, et demandent beaucoup de patience et minutie.

[1] En rusant quelque peu, il est possible d'écrire à la main un code assembleur évitant ce *spill*, mais ceci est laissé comme exercice au lecteur. Dans nos tests, ni GCC ni LLVM ne put éviter cela.

[2] https://camel-cdr.github.io/rvv-bench-results/articles/vector-utf.html

5.1. Conditions expérimentales

Traditionnellement, une façon rapide d'estimer les performances d'un programme serait d'établir un modèle de coût, et de tenter de modéliser l'exécution de l'algorithme à travers ce modèle. C'est ce que fait le programme `llvm-mca`, qui utilise une modèle de coût prenant en compte différents aspects des processeurs modernes pour tenter de modéliser au mieux l'exécution d'un bout de code assembleur donné en entrée. Cependant, en ce qui concerne RVV 1.0, les modèles que contient `llvm-mca` sont souvent trop pessimistes, et souvent utilisés en « boîtes noires », car ils proviennent de processeurs aux designs propriétaires. De plus, ces modèles reposent sur des configurations très spécifiques de processeurs, ce qui fait qu'il n'est pas possible de mesurer l'impact de différents paramètres (e.g. latence mémoire, taille de vecteur, …) sur l'exécution.

Par conséquent, nous utilisons Vicuna[1] pour mesurer le nombre de cycles requis pour l'exécution d'un algorithme, en mesurant à la fois l'implémentation scalaire et vectorielle sur les mêmes données. On spécifie le programme binaire à exécuter lors du lancement de la simulation, puis le programme indique à travers le « port UART » simulé le nombre de cycles écoulés pour chaque implémentation.

```
$ make run-mask_vv
scalar: 31761 cycles / 18862 instructions
vector: 5436  cycles / 1520 instructions
```

Fig. 5. – Exemple de résultat de l'exécution d'une variante de `mask`

Dans ces conditions expérimentales, il n'y a pas d'interférence d'un quelconque système d'exploitation ni de processus concurrents. Puisque le processeur est réinitialisé à chaque simulation, nous n'avons pas non plus d'interférence entre les *runs*. On est donc dans une situation idéale pour mesurer uniquement la durée d'exécution du programme.

Vicuna est configuré en mode « compact » : registres vectoriels de 128 bits, bus mémoire de 32 bits, et pipeline partagée pour toutes les unités d'exécutions.

Chaque programme traite des chaînes d'environ un millier de caractères stockées sur la pile.

5.2. Comparaison des implémentations scalaires et vectorielles

En collectant ces résultats pour les différents algorithmes, on obtient la fig. 6.

On peut remarquer qu'il y a une large différence entre les versions scalaires et vectorielles : par exemple, la variante `memchr` utilisant RVV est environ 4x plus rapide que son équivalent séquentiel. Ceci est cohérent avec les accélérations typiques trouvées lors du montage du projet sur les versions AVX de nos programmes (sur une machine de bureau standard).

Notons que la limite théorique de performance dépend de la taille des vecteurs, et serait atteinte si le traitement de chaque vecteur prenait autant de temps que le traitement d'un élément dans l'algorithme scalaire. Par exemple, dans le cas de `memchr`, la limite théorique de l'accélération est 16x par rapport à la base scalaire (avec la configuration Vicuna utilisé pour les mesures fig. 6).

[1]Voir section 3.1

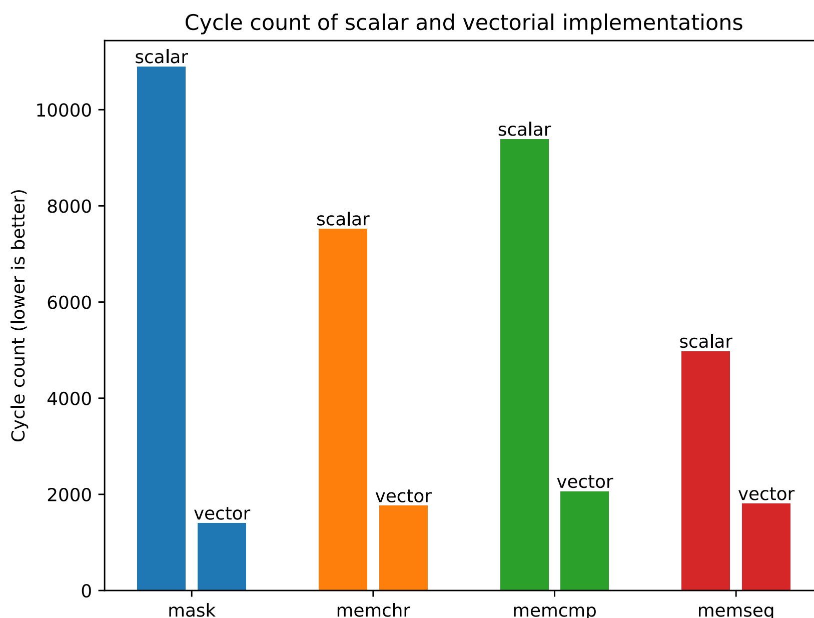

Fig. 6. – Nombre de cycles utilisés lors de l'exécution des implémentations scalaires et vectorielles des différents algorithmes. Un nombre de cycles plus petit indique un temps d'exécution plus court.

5.3. Comparaison de variantes vectorielles

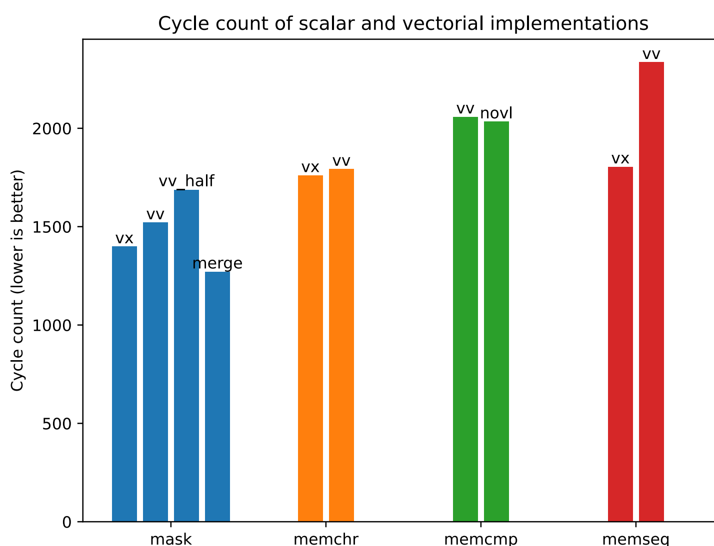

Fig. 7. – Nombre de cycles écoulés lors de l'exécution de différentes variantes de chaque algorithme. Un plus petit nombre de cycle indique un temps d'exécution plus court.

De plus, nous avons pu mesurer les différences de performances entre les variantes écrites dans la section 4.4. Sur la fig. 7, on peut observer que les variantes effectuant des opérations *vecteur-à-registre* (notées `vx`) ont généralement un petit gain de performance par rapport à celles utilisant des opérations *vecteur-à-vecteur* (notées `vv`), entre 2% (`memchr`) et 25% (`memseq`).

Ces mesures sont particulièrement intéressantes dans le cadre du projet **SxC**, car elles permettent d'informer les choix de conception du back-end du compilateur de *Vectoid*.

6. Conclusion

Dans le cadre du projet **SxC**, ce stage aura permis d'explorer à la fois l'état du support pour l'extension vectorielle RISC-V, mais aussi les possibilités et challenges offerts par celle-ci. Il aura également aidé à poser les bases d'une chaîne de développement automatisée permettant le débuggage, la vérification fonctionnelle, et la mesure de performance de programmes RVV.

Du côté personnel, ce stage m'a permis de me pencher sur des domaines qui m'intéressaient depuis longtemps, mais pour lesquels je n'avais jamais vraiment eu l'occasion ou le temps de m'y plonger en aussi grand détail. En plus de cela, j'ai touché à beaucoup plus de technologies que je ne pensais originellement, notamment Verilog (le langage de conception qu'utilise Vicuna) et Verilator (le simulateur Verilog utilisé).

Cependant, une fois mon stage fini, j'avais encore quelques tâches encore en cours (comme par exemple l'automatisation des benchmarks pour une configuration donnée, ou le relevé de la latence (individuelle) des instructions vectorielles sur Vicuna), et bien que celles-ci n'étaient pas forcément requises à la complétion du stage, une partie de moi est toujours insatisfaite de n'avoir pas pu les compléter. L'écriture de ce rapport a occupé grande partie de la fin du stage, plus grande que je ne m'y attendais, et a déraillé les échéances que j'avais prévu pour ces tâches. Ce n'est pas la première fois que la rédaction se révèle être extrêmement chronophage pour moi, et c'est un domaine sur lequel je dois absolument m'améliorer si je veux pouvoir continuer dans cette voie.

Bibliographie

[PP21] M. Platzer et P. Puschner, « Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation », in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, B. B. Brandenburg, Éd., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 196. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, p. 1-18. doi: 10.4230/LIPIcs.ECRTS.2021.1.

[Per+22] M. Perotti, M. Cavalcante, N. Wistoff, R. Andri, L. Cavigelli, et L. Benini, « A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design », in *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2022, p. 43-51. doi: 10.1109/ASAP54787.2022.00017.

[Shi+23] M.-S. Shih, H.-M. Lai, C.-L. Lee, C.-K. Chen, et J.-K. Lee, « Register-Pressure Aware Predicator for Length Multiplier of RVV », in *Workshop Proceedings of the 51st International Conference on Parallel Processing*, in ICPP Workshops '22. Bordeaux, France: Association for Computing Machinery, 2023. doi: 10.1145/3547276.3548513.

A. Annexes

A.1. Détails des macros utilisées dans le code RVV

Le type `vu8m8_t` représente un vecteur RVV contenant des entiers non-signés de 8 bits, et qui a une multiplicité de 8 (c'est-à-dire qui occupe 8 registres vectoriels). Le type `vbool1_t` représente un masque résultant d'une opération sur un `vu8m8_t`.

Nom	Instruction RVV	Sémantique
`size_t` **`vsetvl_e8m8`**`(size_t len)`	`vsetvli res, len, e8, m8`	Signale que l'on traite des éléments de 8 bits, avec 8 registres par vecteurs. `res = vl = min(len, VLEN/8*8)`
`size_t` **`vsetvlmax_e8m8`**`()`	`vsetvli res, x0, e8, m8`	**`vsetvl_e8m8`**`(VLEN/8*8)`
`vu8m8_t` **`vle8`**`(char *addr)`	`vle8.v res, (addr)`	`res[i] = addr[i]`
`vu8m8_t` **`vmv_v_x`**`(char c)`	`vmv.v.x res, c`	`res[i] = c`
`void` **`vse8_m`**`(vbool1_t mask, char *dst, vu8m8_t vs)`	`vse8.v vs, (dst), mask`	`if mask[i] then dst[i] = vs[i]`
`void` **`vse8`**`(char *dst, vu8m8_t vs)`	`vse8.v vs, (dst)`	`dst[i] = vs[i]`
`vu8m8_t` **`vslide1up_vx`**`(vu8m8_t vs, char c)`	`vslide1up.vx res, vs, c`	`res[i] = vs[i-1]; res[0] = c`
`vu8m8_t` **`vnot`**`(vu8m8_t vs)`	`vnot.v res, vs`	`res[i] = ~vs[i]`
`vbool1_t` **`vmseq_vx`**`(vu8m8_t vs, char c)`	`vmseq.vx res, vs, c`	`res.bits[i] = (vs[i] == c ? 1 : 0)`
`vbool1_t` **`vmseq_vv`**`(vu8m8_t v1, vu8m8_t v2)`	`vmseq.vv res, vs1, vs2`	`res.bits[i] = (vs1[i] == vs2[i])`
`vbool1_t` **`vmsne_vv`**`(vu8m8_t v1, vu8m8_t v2)`	`vmsne.vv res, vs1, vs2`	`res.bits[i] = (vs1[i] != vs2[i])`
`long` **`vfirst`**`(vu8m8_t vs)`	`vfirst.m res, vs`	`res = indexof(vs.bits, 1)`
`vbool1_t` **`vmand`**`(vbool1_t a, vbool1_t b)`	`vmand.mm res, a, b`	`res.bits[i] = a.bits[i] & b.bits[i]`
`vbool1_t` **`vmnot`**`(vboolt1_t a)`	`vnot.m res, a`	`res.bits[i] = ~a.bits[i]`

Tableau 1. – Correspondance entre macro et instruction RVV.
Pour la colonne « **Sémantique** », $0 \leq i < \text{vl}$.

Les fonctions représentées dans la première colonne du tableau correspondent en réalité à des *intrinsèques*, qui sont des fonctions spéciales permettant de modéliser une instruction spécifique comme si c'était une simple fonction C classique. Ceci nous permet d'éviter d'écrire du code assembleur directement (avec tous les détails additionnels que cela expose), mais permet tout de même d'émettre une instruction spécifique.

A.2. Détails des macros utilisées dans le code AVX2

La macro `vector_size` correspond au nombre d'octet dans un vecteur; en AVX2, chaque vecteur fait 256 bits, soit 32 octets, donc `vector_size = 32`. Le type `vector` correspond à un vecteur AVX2, tandis que `mask` est simplement un `uint64_t` (`long`), qui contient autant de bits qu'il y a d'octets dans un vecteur.

Nom	Instruction AVX2	Sémantique
`vector load_mask(char c)`	`vpbroadcastb res, c`	`res[i] = c`
`vector uload(void* addr)`	`vmovdqu res, addr`	`res[i] = addr[i]`
`vector vectorEq(vector a, vector b)`	`vpcmpeqb res, a, b`	`res[i] = (a[i] == b[i] ? 1 : 0)`
`mask maskFromVector(vector a)`	`vpmovmskb res, a`	`res.bits[i] = (a[i] == 1 ? 1 : 0)`

Tableau 2. – Correspondance entre macro et instruction AVX2.
Pour la colonne « **Sémantique** », $0 \leq i < \text{vector_size}$.

Comme avec RVV, les « fonctions » présentées sont en réalité des intrinsèques.

A.3. Implémentations RVV des algorithmes restants

Cette section détaille brièvement la spécification et l'implémentation des différents algorithmes non traités dans le reste du rapport.

A.3.1. Implémentation de `mask`

L'algorithme `mask` prend en entrée deux tableaux `src` et `dst`, ainsi qu'une valeur de référence `c`, et écrit dans `dst` la valeur `1` aux emplacements dans `src` contenant `c`, et `0` autrement.

mask (src, dst, c, N):
1 **for** $i \leftarrow 0$ **to** $N - 1$
2   **if** src$[i] = c$ **then**
3   dst$[i]$ = 1
4   **else**
5   dst$[i]$ = 0

Fig. 8. – Pseudo-code pour **mask**

```c
char *mask(char *src, char *dst, char c, size_t length) {
  char *currSrc = src;
  char *currDst = dst;

  // initialize two vectors full of 1s and 0s
  size_t max_vl = vsetvlmax_e8m8();
  vu8m8_t ones_vec   = vmv_v_x(1);
  vu8m8_t zeroes_vec = vmv_v_x(0);

  size_t dataLeft = length;
  while (dataLeft > 0) {
    // declare that we're trying to process `n` elements,
    // each of size 8 bits; we also group vectors by 8,
    // so that we can load as many elements as possible
    size_t vl = vsetvl_e8m8(dataLeft);
    // load 8-bits elements from `currSrc`
    vu8m8_t vec_src = vle8(currSrc);
    // create a bitmask with '1's where the elements where equal, and '0' if not
    vbool1_t dst_mask = vmseq_vx(vec_src, c);
    // use the mask to write the '1's
    vse8_m(dst_mask, currDst, ones_vec);

    // use the *opposite* of the mask to store the '0's
    vbool1_t inverted_mask = vmnot(dst_mask);
    vse8_m(inverted_mask, currDst, zeroes_vec);

    // advance the streams
    currSrc += vl;
    currDst += vl;
    dataLeft -= vl;
  }

  return dst;
}
```

Listing 8. – Implémentation RVV de `mask`

A.3.2. Implémentation de `memcmp`

L'algorithme `memcmp` prend en entrée deux tableaux `a` et `b`, et renvoie la différence entre leur deux premiers éléments non-égaux, ou 0 si tous leurs items sont identiques.

memcmp (a, b, N):
1 **for** $i \leftarrow 0$ **to** $N - 1$
2   **if** $a[i] \neq b[i]$ **then**
3   **return** $a[i] - b[i]$
4 **return** 0

Fig. 9. – Pseudo-code pour **memcmp**

```c
int memcmp(char *a, char *b, size_t length) {
  // Every iteration, we load a chunk of `a` and a chunk of `b`,
  // and then check for inequalities (`vmsne`+`vfirst`). If there's any
  // mismatches, we get their index (`vfirst`) and return the difference

  size_t dataLeft = length;
  char *curr_a = a;
  char *curr_b = b;

  while (dataLeft > 0) {
    size_t vl = vsetvl_e8m8(dataLeft);

    vu8m8_t chunk_a = vle8(curr_a);
    vu8m8_t chunk_b = vle8(curr_b);

    vbool1_t mismatches = vmsne_vv(chunk_a, chunk_b);

    long mismatch_idx = vfirst(mismatches);
    if (mismatch_idx >= 0)
      return curr_a[mismatch_idx] - curr_b[mismatch_idx];

    curr_a += vl;
    curr_b += vl;
    dataLeft -= vl;
  }

  return 0;
}
```

Listing 9. – Implémentation RVV de `memcmp`

A.4. Exemple d’exécution de différents algorithmes

A.4.1. Exécution de l’implémentation AVX de `memseq`

```
a_mask: load_mask(a: 'e')
        0        1        2        3        28       29       30       31
    +--------+--------+--------+------.....-------+--------+--------+--------+
    |   e    |   e    |   e    |   e  .....  e    |   e    |   e    |   e    |
    +--------+--------+--------+------.....-------+--------+--------+--------+

b_mask: load_mask(b: 'l')
        0        1        2        3        28       29       30       31
    +--------+--------+--------+------.....-------+--------+--------+--------+
    |   l    |   l    |   l    |   l  .....  l    |   l    |   l    |   l    |
    +--------+--------+--------+------.....-------+--------+--------+--------+

chunk: uload(str: "hello john, how are you? how's the job? how are the kids?")
        0        1        2        3        28       29       30       31
    +--------+--------+--------+------.....-------+--------+--------+--------+
    |   h    |   e    |   l    |   l  .....  '    |   s    |        |   t    |
    +--------+--------+--------+------.....-------+--------+--------+--------+

a_in_chunk: vectorEq(chunk, a_mask)
        0        1        2        3        28       29       30       31
    +--------+--------+--------+------.....-------+--------+--------+--------+
    |   0    |   1    |   0    |   0  .....  0    |   0    |   0    |   0    |
    +--------+--------+--------+------.....-------+--------+--------+--------+

b_in_chunk: vectorEq(chunk, b_mask)
        0        1        2        3        28       29       30       31
    +--------+--------+--------+------.....-------+--------+--------+--------+
    |   0    |   0    |   1    |   1  .....  0    |   0    |   0    |   0    |
    +--------+--------+--------+------.....-------+--------+--------+--------+

offsets_a: maskFromVector(a_in_chunk)
                         01000000000000000010000000000000

offsets_b: maskFromVector(b_in_chunk)
                         00110000000000000000000000000000

result = (offsets_a << 1) & offsets_b
                         00100000000000000001000000000000
                       & 00110000000000000000000000000000
                       = 00100000000000000000000000000000
                         ^                              ^
                         |                              |
                       bit 0                         bit 31
```

Listing 10. – Exemple d’exécution de l’implémentation AVX `memseq` (listing 5)