Rapport de stage

Stage du 16 mai 2022 au 15 juillet 2022

GOEDEFROIT Charles

16 juillet 2022

Table des matières

1 Contexte **2**

2 Objet du stage / travail demandé **2**
2.1 Les bibliothèques 2
2.2 Les objectifs 3

3 Travail réalisé **3**
3.1 Défrichage / démarrage ? 3
3.2 Les types représentent les vecteurs 4
3.2.1 La représentation de vecteur dans *Eigen* 4
3.2.2 La représentation de vecteur dans *MIPP* 4
3.3 Ajout d'une nouvelle architecture *MIPP* 5
3.4 Types et opérations *Eigen* ? 5
3.5 Les tests 5
3.6 L'implémentation 6

4 Conclusion **7**
4.1 Ce que j'ai remarqué 7
4.2 Les connaissances que j'ai acquises lors de mes études et que j'ai utilisées au cours du stage 8
4.3 Ce que m'a apporté le stage 8

5 Futur / ce qui reste à faire **9**

6 Remerciements **9**

1 Contexte

J'ai effectué mon stage dans l'équipe-projet STORM à Inria (Institut National de Recherche en Informatique et en Automatique). Inria est un établissement public à caractère scientifique et technologique. L'équipe STORM (STatic Optimizations and Runtime Methods) travaille dans le domaine du calcul haute performance, le HPC (High Performance Computing). Plus précisément sur de nouvelles interfaces de programmation et langages pour exprimer le parallélisme hétérogène et massif. Le but est de fournir des abstractions des architectures tous en garantissant la compatibilité haute performance aussi qu'une bonne efficacité de calcul et énergétique. L'équipe est constituée de chercheurs, d'enseignants-chercheurs, d'ingénieurs de recherche, de doctorants, et de stagiaires. L'équipe à une culture informatique liée à son domaine de recherche :

- Langues de haut niveau spécifiques à un domaine. (High level domain specific languages)
- Les Runtime hétérogènes, les plates-formes multi-cœurs. (Runtime systems for heterogeneous, manycore platforms)
- Des outils d'analyse et de retour de performance. (Analysis and performance feedback tools)

Les membres permanents ont tous fait des études en informatique et ont fait une thèse en informatique dans le domaine du HPC. Ils ont des compétences diverses : compilation, runtime, architecture matérielle, langage bas niveau, langages parallèles, gestion de tâches, ordonnancement...

Le matériel et les logiciels mis à ma disposition sont un ordinateur portable avec Linux et la possibilité d'installer les applications dont j'ai besoin (vscode, LaTeX...), un accès à la plateforme de calcul *PlaFRIM* qui fournit un ensemble de machines (nœuds) aux chercheurs, entreprise SME et étudiants qui en ont besoin. *PlaFRIM* comporte une multitude de nœuds avec des architectures différentes (SSE, AVX, ARM NEON...).

2 Objet du stage / travail demandé

L'objectif de mon stage est de faire le portage de *Eigen* sur *MIPP* .

2.1 Les bibliothèques

Eigen est une bibliothèque open source écrite en C++11 très utilisé. Elle permet de faire de l'algèbre linéaire, de la manipulation de matrices, de vecteurs, de solveurs numérique et related algorithms. Pour accélérer les calculs, Eigen utilise une vectorisation explicite. Il y a donc une implémentation par architecture. Elle est principalement développée au centre Inria de l'université de Bordeaux et est au cœur d'autres bibliothèques comme TensorFlow.

MIPP est une bibliothèque open source écrite en C++11 qui fournit une abstraction unique pour les fonctions intrinsic [1] (SIMD [2]) de plusieurs architectures. Elle fonctionne actuellement pour les architectures SSE, AVX, AVX512, ARM NEON (32bits and 64bits). Elle supporte les nombres flottants de précision simple et double ainsi que les entiers signés et codés sur 64, 32, 16 et 8 bits. Son objectif est d'écrire une seule fois un code qui utilise les fonctions de *MIPP* , j'appellerai ce code *code MIPP* dans la suite du rapport, sans avoir à écrire un code d'intrinsic spécifique pour chaque architecture. *MIPP* fournit automatiquement, à partir d'un *code MIPP*, les bonnes intrinsic pour une architecture spécifique. *MIPP* est une sous partie d'*AFF3CT* (A Fast Forward Error Correction Toolbox) qui est une bibliothèque et un simulateur qui est dédié au *Forward Error Correction (FEC or channel coding)*. Elle est également écrite en C++.

2.2 Les objectifs

Le premier objectif est donc d'ajouter une nouvelle implémentation vectorielle, en *code MIPP*, des fonctions élémentaires de *Eigen* . Ceci afin de permettre que le support des différentes architectures soit automatique.

Le second objectif est de faire une campagne d'évaluation des performances pour voir s'il y a une différence entre *Eigen* sans l'implémentation en *code MIPP* et *Eigen* avec l'implémentation en *code MIPP*. Il n'y a pas de raison d'avoir de meilleures performances, mais il peut y avoir une légère dégradation.

Le dernier objectif est de tester *Eigen* sur l'architecture *Risc-V* qui n'est pas encore présent dans *Eigen* et évaluer les performances sur simulateur.

L'objectif à long terme est de pouvoir garder uniquement l'implémentation en *code MIPP* qui remplace les autres implémentations explicites.

Les intérêts de ce portage sont :

— la réduction du nombre de lignes de code et de la complexité du code.

— permettre le support de futures architectures sans avoir à refaire toute une implémentation explicite, mais tous simplement en mettant à jour *MIPP* .

3 Travail réalisé

3.1 Défrichage / démarrage ?

Pour prendre en main *Eigen* et *MIPP* j'ai commencé par implémenter un produit vecteur matrice avec les deux bibliothèques.

1. intrinsic : une instruction SIMD
2. SIMD (Single Instruction on Multiple Data) est une architecture parallèle qui permet à une intrinsic de faire simultanément des opérations sur plusieurs données (un ou plusieurs vecteurs) et produire plusieurs résultats

Ensuite j'ai cherché ou se trouvait les implémentations vectorielles explicites dans *Eigen* et comment elles sont implémentées.

Chaque architecture est implémentée en 4 fichiers. Le premier *PacketMath.h* définit les types que *Eigen* utilise pour représenter les vecteurs et les implémentations des opérations vectorielles pour chacun d'entre eux. Il y a une implémentation générique de ce fichier pour les types scalaires. Le second *TypeCasting.h* définit les conversions entre les différents types vectoriel d'*Eigen* . Ces conversions concernent un peu toute les combinaisons de vecteurs de type entier, flottant, double, booléen, half [3] et bfloat16 [4]. Le troisième *MathFunctions.h* implémente les opérations mathématiques non élémentaires comme : `log, log2, log1p, expm1, exp, sin, cos, sqrt, rsqrt, reciprocal, tanh, frexp, ldexp`. Le quatrième *Complex.h* définit les types vectoriels d'*Eigen* et les opérations sur les nombres complexes, simples et doubles précisions.

3.2 Les types représentent les vecteurs

Eigen et *MIPP* représentent les vecteurs différemment, voyons cela.

3.2.1 La représentation de vecteur dans *Eigen*

Les vecteurs sont de taille fixe et représentés par des types. Ces représentations sont dépendantes de l'architecture visée. Ces types sont nommés *Packet* suivis d'un nombre. Ce nombre correspond au nombre d'éléments que le vecteur peut contenir. Les types se terminent par une ou plusieurs lettres qui correspondent au type des éléments du vecteur. Par exemple, un vecteur qui contient quatre flottants sera nommé *Packet4f* et correspondra au type AVX *__m128* car un flottant est codé sur 32 bits donc $32 \cdot 4 = 128$. Il y a donc des représentations de différentes tailles pour les entiers, les flottants (simple et double précision), les nombres complexes... *Eigen* et donc capable d'utiliser des tailles de vecteur différentes en même temps.

3.2.2 La représentation de vecteur dans *MIPP*

Les vecteurs sont de tailles variables en fonction de l'architecture visée. *MIPP* permet d'obtenir la taille des vecteurs qu'il manipule. Le *code MIPP* doit respecter le fait que les vecteurs ont une taille variable. Il n'y a donc que 2 types de vecteurs : *reg* et *reg2* qui fait la moitié de la taille de *reg*. *MIPP* n'est donc pas capable d'utiliser différentes tailles de vecteur à la fois.

3. type présent dans *Eigen*
4. le type bfloat16 est décrit ici. Il y a une implémentation présente dans *Eigen*

3.3 Ajout d'une nouvelle architecture *MIPP*

Pour ajouter la nouvelle architecture *MIPP* , j'ai créé un nouveau dossier `MIPP` dans le dossier `Eigen/src/Core/arch/`. J'ai ensuite créé les 4 fichiers *PacketMath.h*, *TypeCasting.h*, *MathFunctions.h* et *Complex.h* en copiant le contenu des fichiers des architectures *SSE*, *AVX* et *AVX512* que j'ai adapté pour que tout fonctionne. Je me suis basé sur ces trois architectures, car se sont celles qui se trouvent sur mon ordinateur. Il a été aisé de les faire fonctionner ensemble, car lorsque l'architecture *AVX* est définie, l'architecture *SSE* l'est aussi. Une fois l'architecture ajoutée, j'ai modifié le fichier `Eigen/src/Core/util/ConfigureVectorization.h` pour y ajouter la définition de la macro *EIGEN_VECTORIZE_MIPP* dans le cas où *___MIPP___* est défini à la compilation. Cette macro se définit en plus des macros de l'architecture actuelle *EIGEN_VECTORIZE_AVX*, *EIGEN_VECTORIZE_SSE*... Pour finir, j'ai modifié le fichier `Eigen/Core` pour charger les quatre fichiers de la nouvelle architecture *MIPP* . Le chargement de l'architecture *MIPP* est prioritaire par rapport aux autres.

3.4 Types et opérations *Eigen* ?

J'ai listé les différents types *Eigen* définis ainsi que leurs opérations pour les architectures *SSE*, *AVX*, *AVX2* et *AVX512*.

Les types :

SSE `__m128`	**AVX** `__m256`	**AVX2** `__m256`	**AVX512** `__m512`
Packet4f	Packet8f	Packet4l	Packet16f
Packet2d	Packet4d		Packet8d
Packet4i	Packet8i		Packet16i
Packet16b	Packet8h		Packet16h
	Packet8bf		Packet16bf

TABLE 1 – Les types vectoriels *Eigen* par architecture

Les opérations :
// TODO : (tableau avec toutes les fonctions) ?

3.5 Les tests

Avant de commencer l'implémentation en *code MIPP*, j'ai lancé les tests *Eigen* . En lançant ces tests, j'ai remarqué qu'ils sont très long à compiler et à se lancer, de plus, ils ne fonctionnent pas tous à tous les coups. Les tests vont un peu plus vite sur une machine plus puissante, mais cela reste très long.

Pour que je puisse tester efficacement et rapidement, j'ai donc implémenté des tests de non-régression. Pour ces tests, j'ai copié les fonctions

actuelles dans un nouveau fichier et j'ai suffixé ces fonctions par *_old*.

J'ai ensuite implémenté les tests de non-régression pour toutes les opérations présentes dans *PacketMath.h*. Ces tests fonctionnent pour les architectures *SSE*, *AVX*, *AVX2* et quelque opérations en *AVX512*.

Mes tests de non-régression sont capables d'afficher le contenu des vecteurs lorsqu'il y a une différence entre le résultat de la nouvelle version et celui de l'ancienne version de l'opération. Ils sont aussi capables d'afficher le contenu des vecteurs en binaire et de dire, sans arrêter les tests, quand le *code MIPP* appelle une fonction *MIPP* sur un type qui n'est pas encore supporté par *MIPP* .

3.6 L'implémentation

Dans un premier temps, j'ai implémenté seulement les opérations *SSE*, *AVX* et *AVX2*. J'ai commencé par l'implémentation de l'opération *pset1* qui remplie un vecteur avec la même valeur à chaque case. Dans le tableau des types *Eigen* , on voit que les types *SSE* font 128 bits et les types *AVX* font 256 bits. Je suis donc parti sur une mauvaise piste en convertissant les *Packet4f*, *Packet2d* et *Packet4i* en *reg2* et les *Packet8f*, *Packet4d* et *Packet8i* en *reg*. Le problème est que *MIPP* ne support qu'une seule taille de vecteur à la fois donc il ne peut pas faire des opérations sur les *reg2* mais que sur les *reg*. Ce problème m'a amené à devoir transformer mes vecteurs *reg2* en *reg* ce qui se fait avec les fonctions *combinate()* et *low()*, pour respectivement fusionner deux *reg2* vecteurs en un vecteur *reg* et récupérer la première moitié *reg2* d'un vecteur *reg*. Toutes ces transformations amenaient à deux principaux problèmes : le premier est qu'on a un surcoût supplémentaire à chaque appel. Le second est que cette stratégie ne fonctionne pas avec *AVX512* qu'on a mis de côté pour l'instant. Néanmoins, ces premières implémentations fonctionnent et passent mes tests.

Pour palier ce problème, j'ai donc ajouté un système de conversion. Pour cela j'ai dû, pour chaque architecture, ajouter des casts [5] simples et des casts en intrinsic pour les changements de taille. Par exemple, un vecteur de 128 bits que je dois convertir en vecteur de 512 bits et inversement. Ces conversions peuvent amener à une perte de performance.

J'ai ensuite continué l'implémentation en *code MIPP*, de plusieurs opérations pour chaque type *SSE* et *AVX*, qu'elle supporte. Ce qui m'a amené à avoir plusieurs fois la même implémentation *MIPP* avec seulement le type qui change. Par exemple, l'opération *pset1<Packet4f>()* et l'opération *pset1<Packet4i>*.

Dans un second temps, le but à été de *replier* les fonctions pour en avoir qu'une seule. Mais je suis tombé sur un problème, le comportement par défaut est une opération scalaire. En effet, *Eigen* permet d'activer ou non

5. conversion d'un type à un autre

la vectorisation. Lorsque la vectorisation et désactivée, ce sont les mêmes opérations qui sont appelées, mais dans ce cas, ce sont les opérations par défaut qui sont appelées. Et dans le cas où la vectorisation est activée, on tombe sur un cas spécifique, les cas avec les *Packet* qui sont précisés à l'appel. On se retrouve donc avec 2 cas génériques, un pour les scalaires et un pour les vecteurs (*Packet*). Cette situation amenait à un état compliqué qui reste encore à résoudre.

Je vois deux solutions qui fonctionneraient peut-être :

— finir de tout implémenter puis faire en sorte que tous les *Packet* correspondent au type *reg* de *MIPP* ce qui permettrait de n'avoir qu'un seul cas spécifique qui correspondrait au *code MIPP*. Cela nous permettrait en plus de ne plus avoir besoin du système de conversion.

— ajouter des adaptateurs d'objet pour avoir une arborescence et utiliser le polymorphisme pour avoir un seul cas spécifique qui correspondrait au *code MIPP*.

4 Conclusion

Le premier objectif est en partie rempli, car j'ai implémenté une partie des opérations et j'ai bien vu les différences entre la manipulation des vecteurs dans *Eigen* et *MIPP* . Par-contre je n'ai pas pu faire une campagne d'évaluation des performances, ni le test d'*Eigen* sur l'architecture *Risc-V*, car toutes les opérations ne sont pas codées en *code MIPP*.

4.1 Ce que j'ai remarqué

— Les tests unitaires d'*Eigen* sont très longs ce qui fait qu'ils ne m'ont pas été utiles pour tester mes modifications.

— Les tests unitaires d'*Eigen* ont des bugs ce qui fait qu'il faut les lancer plusieurs fois pour qu'ils fonctionnent.

— J'ai remarqué que plus je codais en *code MIPP* dans *Eigen* , plus le temps de compilation était élevé. C'est très certainement dû aux templates mais je ne peux pas dire si cela vient des templates dans *MIPP* ou dans *Eigen* ou les deux.

Grâce à mes tests et mes implémentations *MIPP* dans *Eigen* j'ai pu lister des opérations qui manquent dans *MIPP* :

× : absent de *MIPP* ✓ : présent dans *MIPP*

	AVX512	**AVX2**	**AVX**	**SSE4.1/4.2**	**SSE2/3**
add<int32_t>	✓	✓	×	✓	✓
sub<int32_t>	✓	✓	×	✓	✓
mul<int32_t>	✓	✓	×	✓	×
orb<int8_t>	✓	✓	×	✓	✓
xor<int8_t>	✓	✓	×	✓	✓
and<int8_t>	✓	✓	×	✓	✓
cmpneq<int16_t>	×	✓	×	×	×
cmpneq<int8_t>	×	✓	×	×	×

TABLE 2 – Abstractions *MIPP* non implémentées

Pour exécuter le code en *AVX512* j'ai utilisé *PlaFRIM*.

4.2 Les connaissances que j'ai acquises lors de mes études et que j'ai utilisées au cours du stage

— Les tests de non-régression que j'ai dû implémenter pour vérifier que mes implémentations étaient conformes à la version précédente. J'ai obtenu ces compétences lors des cours d'Architecture Logiciel (AL), de Projet de Programmation (PdP) et dans un cours de BTS SIO (SLAM4).
— La factorisation de code et l'utilisation des 5 principes solides que j'ai appris en cours d'Architecture Logiciel (AL) et de Projet de Programmation (PdP).
— L'utilisation des intrinsics et le calcul vectoriel que j'ai vu en Programmation sur Architecture Parallèles (PAP).
— Les bases en C++ que j'ai vu un petit peu dans Nachos en TP de Système d'Exploitation (SE) mais aussi dans le projet PdP de mon équipe.
— La compréhension du monde de la recherche grâce à l'UE Initiation recherche de L3.

4.3 Ce que m'a apporté le stage

— Une meilleure compréhension des casts, des conversions des types de base et de la manipulation des vecteurs.
— Une nouvelle expérience de développement dans un projet open source qui est très utilisé.
— L'utilisation avancée des templates en C++.
— Des utilisations différentes des vecteurs avec plusieurs visions de leur utilisation.

— Une meilleure compréhension du monde de la recherche.

5 Futur / ce qui reste à faire

Il reste à implémenter toutes les opérations qui ne sont pas encore en *code MIPP* et faire un *repliage* qui fonctionne correctement.

Il serait intéressant d'ajouter les évolutions suivantes à *MIPP* :

— Ajouter le type boolean (`bool`), le type half (*HF* ou `_m512h` en *AVX512*) et les nombres complexes.
— Ajouter des types non signés (`uint8_t`, `uint16_t`...)
— Faire en sorte que les *Reg2* soit affichable en implémentant l'opérateur *to string*.
— Ajouter la correspondance int8_t et char pour permettre à l'utilisateur de faire des casts avec char.
— Faire en sorte que l'abstraction `testz` fonctionne sur toutes les architectures.
— Peut-être ajouter les opérations pour le Reg2 mais ce n'est peut-être pas dans l'esprit de *MIPP* .
— Trouver ce qui cause un temps de compilation excessif et le réduire.

En vert les types qui fonctionnent et en orange les types qui ne fonctionnent pas

Type standard	**type de base**	**type unsigned**
int8_t	char	uint8_t
int16_t	short	uint16_t
int32_t	int	uint32_t
int64_t	long	uint64_t
float	—	—
double	—	—
bool	—	—

TABLE 3 – Types compatibles avec *MIPP*

6 Remerciements

// merci