

Rapport de stage

Stage du 16 mai 2022 au 15 juillet 2022

GOEDEFROIT Charles

12 juillet 2022

Table des matières

1 Contexte	2
2 Objet du stage / travail demandé	3
2.1 Les bibliothèques	3
2.2 Les objectifs	3
3 Travail à réaliser	4
3.1 Défrichage / démarrage ?	4
3.2 Les types représentent les vecteurs	5
3.2.1 La représentation de vecteur dans Eigen	5
3.2.2 La représentation de vecteur dans MIPP	5
4 Conclusion	8
5 Future / ce qui reste à faire	8
6 Remerciements	9

1 Contexte

J'ai effectuer 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** (**S**Tatic **O**ptimizations and **R**untime **M**ethods) travaille dans le domain du calcul haute performance, le **HPC** (**H**igh **P**erformance **C**omputing). 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 architecture tous en garantissent la compatibilité haute performance aussi qu'une bonne efficacité de calcul et énergétique. L'équipe est constitué de chercheurs, d'enseignants-chercheurs, d'ingénieurs de recherche, de doctorants, et de stagiaires. L'équipe à une culture informatique lié à sont domain de recherche :

- Langues de haute niveau spécifiques à un domaine. (**H**igh 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 on tous fait des étude en informatique et on eu une thèse en informatique dans le domain du **HPC**. Ils on des compétence diverse : compilation, runtime, architecture materiel, langage bas niveau, langage parallèles, gestion de tâches...

Le materiel et les logiciels mis à ma disposition sont un ordinateur portable avec linux et la possibilité d'installer les application d'on j'ai besoin (vscode, **L**A_TE_X...), un accès à la platforms de calcul *PlaFRIM* qui fournir un ensemble de machine (noeuds) au chercheurs, entreprise **SME** et étudiants qui en on besoin. *PlaFRIM* comporte une multitude de noeuds avec des architectures différentes (**SSE**, **AVX**, **ARM NEON**...).

2 Objet du stage / travaille 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 écrit 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 vectorization explicite. Il y a donc une implémentation par architectures. Elle est principalement développée au centre Inria de l'université de Bordeaux et est au cœur de d'autres bibliothèques comme TensorFlow.

MIPP est une bibliothèque open source écrit en C++11 qui fournit une abstraction unique pour les fonctions intrinsèque¹ (SIMD²) 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 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'intrinsèque spécifique pour chaque architecture. *MIPP* fournit automatiquement à partir d'un *code MIPP* le bon intrinsèque 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

1. intrinsèque : une instruction SIMD

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

fonctions élémentaires de *Eigen*. Ce-ci a fin de permettre que le support des différentes architectures sois automatique.

Le second objectif est de faire une campagne d'évaluation des performances pour voir si 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 mayeur 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 a long terme est de pouvoirs garder uniquement l'implémentation en *code MIPP* qui remplace les autre implémentation explicite.

Les intérêts de ce portage sont :

- la reduction du nombre de lignes de code et de la complexité du code.
- de permettre le support de future architectures sans avoir à refaire tous une implémentation explicite mais tous simplement en mettant a jour *MIPP*.

3 Travail à réaliser

3.1 Défrichage / démarrage ?

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

Ensuite j'ai cherché ou j'ai trouvé 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 le type scalaire. Le second *TypeCasting.h* définit les conversions entre les différents types vectoriels d'*Eigen*. C'est conversions concerne un peu toutes les combinaisons de vecteurs de type entier, flottant, double, booléen, half^3 et bfloat16^4 . Le troisième *MathFunctions.h*

3. type présent dans *Eigen*

4. le type bfloat16 est décrit [ici](#). Il y a une implémentation

implémente les opérations mathématiques non élémentaires comme : `log, log2, loglp, 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, simple et double précision.

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

```
// Des vecteurs à taille fixe qui sont dépendent
de l'architecture visée ...
// Ils sont nommés Packet suivi d'un nombre, le
nombre d'éléments qu'il contient, et qui termine
par une ou plusieurs lettres qui correspondent au
type des éléments dans le vecteur. Par exemple
un vecteur qui contient 4 flottants sera Packet4f.
// Sur les entiers et les flottants de différentes
tailles.
// Eigen est capable d'utiliser plusieurs tailles
de vecteurs à la fois...
```

3.2.2 La représentation de vecteur dans MIPP

```
// Des vecteurs avec des tailles qui sont variables
et indépendantes de l'architecture.
Il y a donc que 2 types de vecteurs reg et reg2
(qui font la moitié de la taille de reg).
// MIPP n'est pas capable d'utiliser plusieurs
tailles de vecteurs à la fois.
// dans 'Eigen/src/Core/util/ConfigureVectorization.h'
j'ai ajouté un cas __MIPP__
// j'ai ajouté une nouvelle architecture MIPP
dans 'Eigen/src/Core/arch/' et que j'ai chargé dans
'Eigen/Core'
// J'ai copié le contenu des architectures SSE,
AVX et AVX512 dans le même fichier. C'est 3
architectures qui fonctionnent ensemble (AVX dépend
de SSE, etc)
// J'ai listé les différentes types Eigen définit
pour chaque architecture aussi que les opérations
supportées (tableau avec toutes les fonctions)
```

présent dans *Eigen*

SSE	AVX	AVX2	AVX512
Packet4f	Packet8f	Packet4l	Packet16f
Packet2d	Packet4d		Packet8d
Packet4i	Packet8i		Packet16i
Packet16b	Packet8h		Packet16h
	Packet8bf		Packet16bf

Table 1 – Les type vectoriel Eigen par architecture

// J'ai lancer les tests Eigen sur mon architecture fraîchement copier de AVX.

// En lançant les tests Eigen j'ai remarquer qu'il sont très long à ce lancer et qu'il ne fonctionne pas à tous les coup.

// Pour que je puis tester efficacement et rapidement j'ai implémenter des tests de non regression. Pour exécuter ces tests j'ai copier les fonctions actuel dans un nouveaux fichier et j'ai suffixés ces fonctions par *_old*.

// J'ai implémenter les test de non regression pour la fonctionne *pset1*, qui rempli un vecteur avec la même valeur a chaque case, et j'ai fait l'implémentation en MIPP. Pour finir j'ai lancer mes tests qui passé.

// J'ai implémenter les test de non regression pour toutes les operations présent dans *Packet-Math.h*. Ces test fonctionne pour les architectures **SSE**, **AVX**, **AVX2** et quelque **AVX512**.

// Mes test de non régression son capable d'afficher le contenu des vecteur dans le cas ou il y a une différence entre la nouvelle version et l'ancienne version de l'operation. Il sont aussi capable d'afficher le contenu des vecteur en binaire et d'afficher sans plantage quant une operations n'existe pas dans MIPP.

// J'ai commencer sur une mauvaise piste con convertissant les Packet Eigen en reg2 lors qu'il été petit. Ce qui fait que je devais faire la transformation, *combinate()*, entre reg et reg2 a chaque fois que je voulais faire un calcule car, dans MIPP, il n'y a presque aucune operations sur les reg2. De plus cela ne fonctionné pas avec AVX512 et ajouter pleins de complication ce qui fait que j'ai du faire des conversion.

// Comme il MIPP ne peut pas utilisé des taille de vecteur différant je suis obliger de cast les vecteur entre Eigen et MIPP...je me retrouve donc à faite des conversion entre des vecteurs de taille différant par exemple un vecteur de 128

bits que je dois convertir en vecteur de 512 bits et inversement. Ces conversion peuvent amener à une perte de performance. (Benchmark here ?)

// Dans un premier temps, pour chaque fonction, j'ai fait l'implémentation en MIPP pour chaque type quelle support. Ce qui m'a amené à avoir 2 fois la même implémentation MIPP avec seulement le type qui change par exemple une *pset1<Packet4f>* et une autre *pset1<Packet4i>*. Dans un second temps le but a été de replier les fonctions pour en avoir qu'une seule. Mais je suis tombés sur un problème, le comportement par défaut et une opération scalaire. En effet Eigen permet d'activer ou non la vectorisation. Lors que la vectorisation est désactivée se sont les mêmes opérations qui sont appelées mais, dans ce cas, c'est l'opération par défaut qui est appelée et quand la vectorisation est activée on tombe sur un cas spécifique (Le Packet) et préciser à l'appeler. On se retrouve donc avec 2 cas spécifiques, un pour les scalaires et un pour les vecteurs. Cette situation amène à un état compliqué qui reste encore à résoudre. Peut-être 2 solutions :

// * Tous coder et faire en sorte que tous les Packet corresponde au type reg de MIPP, ce qui permet d'avoir un seul type.

// * Ajouter des adaptateurs pour avoir une arborescence et utiliser le polymorphisme.

// Grâce à mes tests et mes quelques implémentations j'ai pu lister des opérations qu'il manque dans MIPP :

x : absent de MIPP ✓ : présent dans MIPP

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

Table 2 – Abstractions MIPP non implémentées

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

4 Conclusion

```
// J'ai remarquer que :  
// * Les test unitaire de Eigen sont très long  
ce qui fait qui ne mon pas été utile pour teseter  
mes modifications  
// * Les test unitaire de Eigen on des bug  
ce qu'il fait qu'il faut les lancer plusieurs fois  
pour qu'il fonctionne (c'est un bug connue) (trouver  
l'issus)  
// * J'ai remarquer que plus on code en MIPP  
dans Eigen plus le temps de compilation est  
élever. C'est très certainement du au templates  
mais je ne peut pas dire si cela viens des  
templates dans MIPP ou dans Eigen ou les 2.  
// Les connaissances que j'ai acquis lors de mes  
étude que j'ai utiliser au cours du stage :  
// * Les test de non regression que j'ai du  
implementer pour verifier que mes implémentation  
été bonne. (AL / PdP / PLE / SLAM4) (Le GL)  
// * Le fait de refactorisé du code (utilisation  
des 5 principe...) (AL / PdP)  
// * L'utilisation des intransics et le calcule  
vectoriel (vue en PAP)  
// * Les base en C++ vue un petit-peu en OS  
dans nahos mais aussi dans mon projet PdP...  
// * La comprehension du monde de la recherche  
grace à l'UE initiation recherche.  
// Ce que ma apporter le stage :  
// * une mateur comprehension des cast, conver-  
sion des type de base et des vecteurs  
// * Utilisation avancer des templates en c++  
// * des Utilisation différentes des vecteurs,  
plusieurs vision de leur utilisation. (taille fix ou  
non)  
// * une mateur comprehension du monde de la  
recherche
```

5 Future / ce qui reste à faire

```
// Il reste a implementer tous le reste  
// Faire un repliage qui fonctionne  
// Modifier MIPP pour ajouter tous ce qu'il  
manque...  
// Evolution possible pour MIPP :  
// * Ajout du type bool, d'un type h et complex  
(present en AVX512)
```

```

// * ajouter la correspondance int8_t et char
(pour le cast)
// * ajout des type unsigned (uint8_t, uint16_t...)
// * make Reg2 printable
// * peut-être ajouter les operations pour le
Reg2 mais ce n'ai peut-être pas dans l'esprit de
MIPP
// * faire en sorte que le testz fonctionne avec
toute les architecture
// * voire le temps de compilation

```

En **vert** les type qui fonctionne, En **orange** les type qui ne fonctionne

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 – Type compatible avec MIPP

6 Remerciements

// merci