Rapport de stage

Stage du 16 mai 2022 au 15 juillet 2022

GOEDEFROIT Charles

13 juillet 2022

Table des matières

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 calcule haute performance, le **HPC** (**H**igh Performance **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 calcule 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. (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 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, language bas niveau, language 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, LaTeX...), un accès à la platforms de calcule *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 accéléré les calcules Eigen utilise une vectorization explicite. Il y as donc une implémentation par architectures. Elle est principalement développée au centre Inria de l'université de Bordeaux et est au coeur de d'autre bibliothèque comme TensorFlow.

MIPP est une bibliothèque open source écrit en C++11 qui fourni une abstraction unique pour les fonctions intrinsic[1] (SIMD[2]) de plusieurs architectures. Elle fonctionne actuelle pour les architectures SSE, AVX, AVX512, ARM NEON (32bits and 64bits). Elle supporte les nombres flottants de precision simple et double ainsi que les entier signé codé sur 64, 32, 16 et 8 bits. Sont objectif est d'écrit une seule fois un code qui utilise les fonctions de *MIPP* , j'appellerai ce code *code MIPP* dans la suite du rapport, sans avoir a écrire un code d'intrinsic spécifique pour chaque architectures. *MIPP* fourni automatiquement à partir d'un *code MIPP* les bonne intrinsic pour une architecture specific. *MIPP* est un 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 vectoriel, en *code MIPP*, des

1. intrinsic : une instruction SIMD
2. SIMD (Single Instruction on Multiple Data) est une architecture parallèles qui permet à une intrinsic de fait simultanément des operations 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 degradation.

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 à jour *MIPP* .

3 Travaille réaliser

3.1 Défrichage / démarrage ?

Pour prend en main *Eigen* et *MIPP* j'ai commancer par implementer un produit vecteur matrice avec les deux bibliothèques.

Ensuite j'ai chercher ou ce trouvé les implémentations vectoriel explicite dans *Eigen* et comment elles sont implémentées.

Chaque architectures sont implémentées en 4 fichiers. Le premier *PacketMath.h* définit les types que *Eigen* utilise pour représenté les vecteurs et les implémentations des operations vectoriel pour chacun d'entre eux. Il y a une implémentation générique de ce fichier pour les type scalaire. Le second *TypeCasting.h* définit les conversion entre les différant types vectoriel d'*Eigen* . C'est conversions concerne un peut toute les combinaison de vecteurs de type entier, flottant, double, booléen, half[3] et bfloat16[4]. Le troisièmes *MathFunctions.h*

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

implémente les operations mathématique non élémentaire comme : `log, log2, log1p, expm1, exp, sin, cos, sqrt, rsqrt, reciprocal, tanh, frexp, ldexp`. Le quatrième *Complex.h* définit les types vectoriel d'*Eigen* et les operations sur les nombre complex, simple et double precision.

3.2 Les types représentent les vecteurs

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

3.2.1 La representation de vecteur dans *Eigen*

Les vecteurs sont de taille fix et représenté par des types. Ces représentation sont dépendent de l'architecture visé. Ces types sont nommés *Packet* suivi d'un nombre. Ce nombre corresponde au nombre d'éléments que le vecteur peut contenir. Les type ce termine par une ou plusieurs letter qui corresponde au type des éléments du vecteur. Par example un vecteur qui contient quatre flottant sera nommé *Packet4f* et correspondra au type AVX *__m128* car un flottant et codé sur 32bits donc $32 \cdot 4 = 128$. Il y as donc des représentation de différant tailles pour les entiers, les flottants (simple et double precision), les nombres complex...*Eigen* et donc capable d'utilisé des taille de vecteur différente en même temps.

3.2.2 La representation de vecteur dans *MIPP*

Les vecteurs sont de taille variable en fonction de l'architecture visé. *MIPP* permet d'obtenir la taille des vecteurs qu'il manipule. Le *code MIPP* dois respecté Le fait que les vecteurs on une taille variable. Il y a donc que 2 type de vecteurs *reg* et *reg2* qui fait la moitiés de la taille de *reg*. *MIPP* n'est donc pas capable d'utilisé différente taille de vecteur à la fois.

3.3 Ajoute d'une nouvelle architecture *MIPP*

Pour ajouter la nouvelle architecture *MIPP* j'ai créé un nouveaux dossier `MIPP` dans le dossier

présent dans *Eigen*

Eigen/src/Core/arch/. J'ai ensuite créé le 4 fichiers *Packet-Math.h*, *TypeCasting.h*, *MathFunctions.h* et *Complex.h* en copient le contenu des fichiers des architecture *SSE*, *AVX* et *AVX512* que j'ai adapté pour que tous fonctionne. Je me suis basé sur c'est trois architecture car c'est celles qui ce trouver sur mon ordinateur. Il à été aisé de les faire fonctionné ensemble car lorsque l'architecture *AVX* est definit l'architecture *SSE* l'ai aussi. Une fois l'architecture ajouté j'ai modifier 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 définit à la compilation. Cette macro ce définit en plus des macro de l'architecture actuel *EIGEN_VECTORIZE_AVX*, *EIGEN_VECTORIZE_SSE*...Pour finir j'ai modifier 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 autre.

3.4 Types et operations *Eigen* ?

J'ai listé les différents types *Eigen* définit ainsi que leur operations 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 type vectoriel *Eigen* par architecture

Les opérations :

// **TODO :** (tableau avec toute les fonctions) ?

3.5 Les tests

Avant de commancer l'implémentation en *code MIPP* j'ai lancer les tests *Eigen* . En lancent ces tests j'ai remarquer qu'il sont très long à compiler et à ce lancer, de plus il ne fonctionne pas tous à tout les coups. Les tests vont un peut plus

vite sur une machine plus puissante mais cela reste très long.

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

J'ai ensuite implémenté les tests de non regression pour toutes les operations présent dans *PacketMath.h*. Ces test fonctionne pour les architectures *SSE*, *AVX*, *AVX2* et quelque operations en *AVX512*.

Mes test de non régression son capable d'afficher le contenu des vecteurs lorsque il y a une différence entre le résultat de la nouvelle version et celui de l'ancienne version de l'operation. Ils sont aussi capable d'afficher le contenu des vecteurs en binaire et de dire, sans arrêter les test, quant le *code MIPP* appelé une fonction *MIPP* sur un type qui n'est pas encore supporter par *MIPP* .

3.6 l'implémentation

Dans un premier temps j'ai implémenté seulement les operations *SSE*, *AVX* et *AVX2*. J'ai commancé par l'implémentation de l'opération *pset1* qui rempli un vecteur avec la même valeur à chaque case. Dans le tableau des types *Eigen* on vois que les types *SSE* font 128 bits et les type *AVX* font 256 bits. Je suis donc partie 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 seul taille de vecteur à la fois donc il ne peut pas faire des opérations sur les *reg2* mais que sur les *reg*. Ce problème ma amené à devoir transformer mes vecteurs *reg2* en *reg* ce qui ce fait avec la fonction *combinate*() et *low*(), pour respectivement fusionné deux *reg2* vecteurs en un vecteur *reg* et récupérer la premier moitiés *reg2* d'un vecteur *reg*. Toute c'est transformation amené à deux problème principaux. Le premier est que on a un surcoûts supplémentaire a chaque appelle. Le second est que cette stratégie ne fonctionne pas avec *AVX512* qu'on a mis de côté pour l'instant. Néanmoins ces premier implémentations fonctionne et passe mes tests.

Pour palier ce problème j'ai donc ajouté un

système de conversion. Pour cela j'ai du pour chaque architectures ajouté des cast[5] simple et des cast en intrinsic pour les changement de taille, Par exemple un vecteur de 128 bits que je dois convertir en vecteur de 512 bits et inversement. Ces conversion peuvent amené a une perte de performance.

TODO : (Benchmark here ?)

J'ai ensuite continué l'implémentation en *code MIPP* de plusieurs opérations pour chaque types *SSE* et *AVX* quelle support. Ce qui ma amené à avoir plusieurs fois la même implémentation *MIPP* avec seulement le type qui change. Par exemple l'operation *pset1<Packet4f>()* et l'operation *pset1<Packet4i>*.

Dans un second temps le but à été de *replier* les fonctions pour en avoir qu'une seul. Mais je suis tombés sur un problème, le comportement par défaut est une operation scalaire. En effet *Eigen* permet d'activer ou non la vectorisation. Lorsque la vectorisation et désactiver se sont les même operations qui sont appelé mais dans ce cas se sont les operation par défaut qui sont appelé. Et dans le cas où la vectorisation et activer on tombe sur un cas spécifique, les cas avec les *Packet* qui sont précisé à l'appelle. On ce retrouve donc avec 2 cas générique, un pour les scalaires et un pour les vecteurs (*Packet*). Cette situation amené à un état compliqué qui reste encore à résoudre.

Je vois deux solutions qui fonctionneraient peut-être :

- finir de tous implémenté puis faire en sorte que tous les *Packet* corresponde au type *reg* de *MIPP* ce qui permettrai d'avoir qu'un seul cas spécifique qui correspondrai au *code MIPP*. Cela nous permettrai en plus de ne plus avoir besoin du système de conversion.
- ajouté des adaptateur d'objet pour avoir une arborescence et utiliser le polymorphisme pour avoir un seul cas spécifique qui correspondrai au *code MIPP*.

4 Conclusion

Le premier objectif est en partie rempli car j'ai implémenté une partie des opérations et j'ai

5. conversion d'un type à un autre

bien vue les différences entre la manipulation des vecteurs dans *Eigen* et *MIPP* . Par-contre je n'ai pas pus faire une campagne d'évaluation des performances ni le tester d'*Eigen* sur l'architecture *Risc-V* car toute les operations ne sont pas codé en *code MIPP*.

4.1 Ce que j'ai remarquer

- 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.
- J'ai remarquer que plus je codé en *code MIPP* dans *Eigen* plus le temps de compilation été élevé. 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 deux.

Grâce à mes tests et mes implémentations *MIPP* dans *Eigen* j'ai pu lister des operations qu'il manque dans *MIPP* :

× : absent de *MIPP* ✓ : present 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 implementer

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

4.2 Les connaissances que j'ai acquis lors de mes études que j'ai utilisé au cours du stage

- Les tests de non regression que j'ai du implementer pour verifier que mes implémenta-

tions été conforme à la version précédente. J'ai obtenu c'est compétences de des lors des cours d'Architecture Logiciel (AL), de Projet de Programmation (PdP) et dans un cours de BTS (SLAM4).
- La refactorisation de code et l'utilisation des 5 principe solides. Que j'ai appris en cours d'Architecture Logiciel (AL) et de Projet de Programmation (PdP).
- L'utilisation des intransics et le calcule vectoriel que j'ai vue en Programmation sur Architecture Parallèles (PAP).
- Les base en C++ que j'ai vue un petit-peu dans nachos en tp Système d'Exploitation(SE) mais aussi dans le projet PdP de mon équipe.
- La comprehension du monde de la recherche grâce à l'UE initiation recherche de L3.

4.3 Ce que ma apporter le stage

- Une mayeur comprehension des cast, des conversions des type de base et de la manipulation des des vecteurs.
- Une nouvelle expérience de développement dans un projet opensource qui est très utiliser.
- L'utilisation avancé des templates en c++.
- Des utilisation différentes des vecteurs avec plusieurs vision de leur utilisation.
- Une mayeur 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