

Support de MPI/OpenMP et de la vectorisation dans Verificarlo

Bouton Nicolas, Hery Andrianantenaina, Ali Lakbal, Julien Even

2020

Contents

1	Verificarlo	3
1.1	Vectorisation dans le calcul scientifique	3
1.2	Compilation	4
2	Définitions de certains termes techniques	5
3	Résumé des besoins	6
3.1	Vectorisation	6
4	Objectifs	7
4.1	MPI/OpenMP	7
4.2	Vectorisation	7
5	Organisation	7
5.1	Groupe	7
5.2	Git	8
5.3	Réunion avec l'encadrant	8
5.4	Discord	8
6	Support MPI / OpenMP	8
6.1	Notion de parallélisme	8
6.2	Notions indispensables pour le parallélisme	9
6.2.1	Système à mémoire partagée	9
6.2.2	Système à mémoire distribuée	9
6.2.3	Thread ou flot d'exécution	9
6.2.4	Processus	9
6.2.5	Calcul parallèle	9

6.3	Présentation d'Open MPI	9
6.4	Installation d'open MPI	10
6.5	Configuration	10
6.6	Compilation d'open mpi	10
6.7	Installation	10
6.8	Préparation environnement	10
6.9	Description de communication dans Open MPI	10
6.10	Compilation d'un programme parallèle avec verifcarlo	11
7	Vectorisation	11
7.1	Introduction	11
7.2	Test	11
7.2.1	Bon résultat des opérations vectorielles	12
7.2.2	Appel aux probes vectorielles	13
7.2.3	Utilisation des jeux d'instructions vectorielles suivant l'architecture	13
7.3	Support des vecteurs 512 / 256 bits	15
7.4	Ajout de probes vectorielles	15
7.5	Ajout des fonctions vectorielles dans l'interface	15
7.5.1	Backend ieee	16
7.6	Fonctions vectorielles en mode scalaire dans les backends . . .	16
7.7	Fonctions vectorielles en mode vectoriel dans les backends . .	17
7.7.1	Backend ieee	17
7.7.2	Backend vprec	17
7.7.3	Backend mca	18
7.8	Vérification si au moins un backend utilisé implémente les opérations vectorielles	18
7.9	Compilation	19
7.10	Problèmes rencontrés	19
7.11	Connaissances acquises	20
7.11.1	gdb	20
7.11.2	llvm	20
7.12	Conclusion vectorisation	21
7.12.1	Performances attendues	22
8	Conclusion	22

1 Verificarlo

Verificarlo est un compilateur basé sur clang et llvm. Il permet d'intercepter toutes les opérations flottantes et de les analyser. Ce qui permet dans le cadre du calcul scientifique et de la simulation de pouvoir déboguer, valider et optimiser ces opérations ainsi que leurs formats. Les opérations

flottantes ont un intérêt particulier dans le monde du **HPC** (High Performance Computing). Elles peuvent être source d'erreurs sans pour autant être un problème mathématiques. En effet dans le domaine de l'informatique, les ordinateurs ont une limitation matérielle qui ne leur permet pas d'atteindre une certaine précision. C'est là qu'intervient **Verificarlo**, cet outil permet

par instrumentation des opérations flottantes, de pouvoir déboguer les erreurs, dû à la précision machine. Par exemple de pouvoir modifier la taille de l'exposant ou bien de la mantisse. Ou bien même de pouvoir effectuer des opérations flottantes en ajoutant du bruit au résultat afin de simuler une précision machine plus élevée.

1.1 Vectorisation dans le calcul scientifique

Dans le domaine du calcul scientifique, il est important de faire des codes propres (c'est à dire pas d'erreur à l'exécution) et optimisés. Nous avons

vu que le parallélisme permettait d'optimiser notre code mais ce n'est pas le seul moyen. Dans nos processeurs modernes, des jeux d'instructions vec-

torielles sont apparus. Ils ont une taille de 128, 256 ou 512 bits suivant le jeu d'instruction qui correspond respectivement au jeu d'instruction **sse**, **avx** et **avx512**. Ce dernier n'est disponible que sur certains processeurs d'**Intel**. Ces jeux d'instructions permettent d'effectuer plusieurs opérations en même temps sur le même coeur de calcul. Dans notre cas, nous nous

intéressons uniquement au type **float** et **double** qui ont respectivement une taille de 32 bits et de 64 bits. Nous pouvons donc effectuer 4 addi-

tions en même temps pour le type **float** et le jeu d'instruction **sse**. Ce qui théoriquement multiplie par 4 la vitesse de calcul. D'où l'intérêt particulier que l'on porte au support de la vectorisation dans Verificarlo. Dans le but de pouvoir l'utiliser sur des codes dans des centres de calculs.

1.2 Compilation

Voici le comportement d'un compilateur (Voir la figure 1):

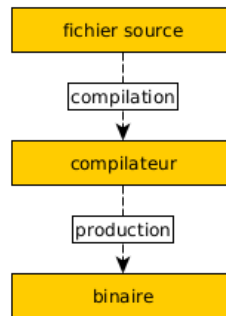


Figure 1: Fonctionnement de base d'un compilateur

Son but est de produire un binaire pour notre architecture à partir d'un fichier source. Le compilateur est généralement décomposé en 3 étapes

(Voir la figure 2):

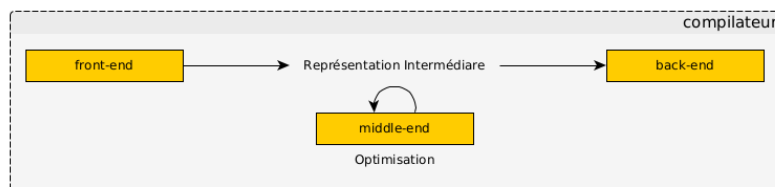


Figure 2: Etape d'un compilateur

Verificarlo est un compilateur. Comme vu précédemment, il suit la décomposition en 3 étapes car il est basé sur **clang** et **llvm**. Les **modifications** qu'apporte Verificarlo se font au **middle-end** avant la phase d'optimisation qui se déroule aussi dans le **middle-end** au niveau de **llvm**. Voici les principales étapes de la compilation avec **Verificarlo** (Voir la fig-

ure 3):

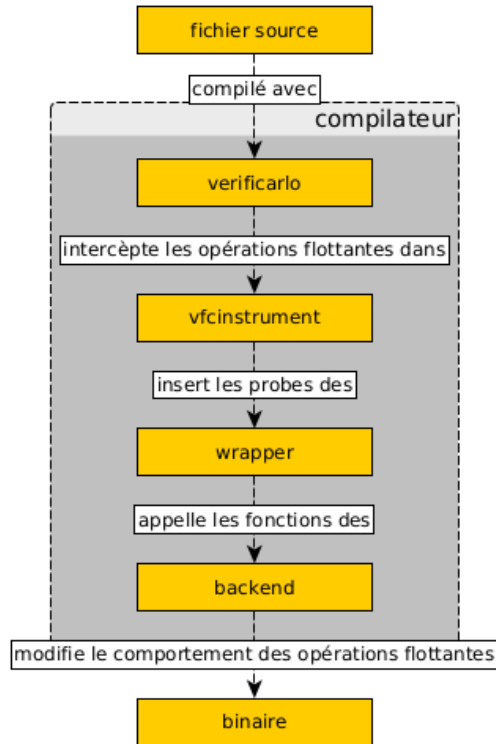


Figure 3: Fonctionnement de Verifcarlo

2 Définitions de certains termes techniques

- **probes:** Les probes sont des fonctions implémenté dans

vfcwrapper qui est linker avec le programme par la partie compilation de veifcarlo.

- **backend:** Dans le cadre de verifcarlo, c'est la/les librairie(s)

dynamique(s) qui seront appelées par le wrapper dans les probes. Dans le cadre d'un compilateur c'est la dernière phase qui descend de la représentation intermédiaire vers le binaire (en général).

- **wrapper:** Ce sont des fonctions qui enveloppent l'appel à d'autres fonctions.
- **link:** Il s'agit de la phase de compilation qui consiste à aller chercher toutes les bibliothèques externes appelées par l'application pour les lier au programme utilisateur afin de résoudre les références non définies.
- **sérialisation:** Dans le contexte de l'utilisation de vecteur il s'agit d'exécuter en séquence les éléments du vecteur.

3 Résumé des besoins

À l'heure actuelle, les codes **multithreadés** (parallèle), particulièrement pour OpenMP, et **vectorisés** nécessitent une sérialisation des opérations ce qui entraîne un surcoût qui peut être évitable. Il faudra donc proposer

des opérateurs vectorisés dans **Verifcarlo** pour les deux outils d'analyse **VPREC** et **MCA**. L'aspect thread-safe de ces opérateurs devra être as-

suré. De plus il faudra générer des nombres aléatoires indépendants pour

l'outil d'analyse **MCA** et la gestion des entrées sorties. Une fois ces out-

ils en place, il sera possible d'étudier l'impact sur la stabilité des codes des environnements MPI et OpenMP. Ainsi que de proposer d'éventuelles analyses.

3.1 Vectorisation

Aujourd'hui **vcinstrument** insère des probes, y compris pour les instructions vectorielles. Celles des instructions vectorielles défont le vecteur et

appellent les versions scalaires au lieu d'appeler les fonctions vectorielles des backends. À l'exécution le **wrapper** charge les bibliothèques dynamiques

(.so) correspondantes au(x) **backend(s)** verifcarlo utilisé (vprec, mca).

4 Objectifs

4.1 MPI/OpenMP

L'objectif ici est de savoir installer et compiler des programmes avec mpi/open en faisant appel à la compilateur verifcarlo.

4.2 Vectorisation

Les changements sont à faire essentiellement dans les **wrappers** et les **backends**:

1. Support des vecteurs de 512 et 256 bits
2. Ajout de **probes** vectorielles appelant les fonctions de **backend** vectorielles
 - Ajout des fonctions vectorielles dans l'interfaces (par pointeurs)
3. Implémenter ces fonctions pour chaque **backend**
 - Faire une première implémentation sérialisé
4. Implémenter la version vectorielle des opérations de base dans le backend **vprec**
 - Prendre en compte les cas spéciaux (dénormaux)
 - Tester la performance sur les NAS (MPI et OpenMP)
5. Faire de même pour le **backend mca**

5 Organisation

5.1 Groupe

Nous nous sommes répartis en 2 groupes:

- un groupe sur la partie MPI / OpenMP ainsi que la génération de nombres aléatoires (Hery Andrianantenaina / Julien Even)
- un groupe sur le support de la vectorisation dans Verifcarlo (Nicolas Bouton / Ali Lakbal)

5.2 Git

Etant donné que **Verificarlo** est un logiciel ayant un dépôt distant sur le site GitHub. Nous avons décidé de créer une Organisation, nommé **Safecarlo** (au passage la plupart des noms sur la méthode Monte Carlo étaient pris et l'aspect **thread-safe** des **wrappers** et des **backends** devait être un des sujets principaux avec la vectorisation), sur **GitHub** et de **fork** Verificarlo dans notre Organisation. Nous avons également chacun **fork** Verificarlo depuis notre Organisation. Voici le lien vers notre **fork** de ver-

ificarlo: Safecarlo Il s'agit de la branche où nous avons réuni les modifi-

cations apportées au cours du projet. Nous avons aussi fait une **pull re-**

quest sur la branche **master** de **Safecarlo** afin que vous puissiez mieux voir les changements apportés. Lien de la pull request

5.3 Réunion avec l'encadrant

Nous avons une réunion toutes les semaines le mardi après-midi avec notre encadrant pour faire le point sur l'avancement de la semaine.

5.4 Discord

Nous nous sommes créé un discord pour pouvoir échanger entre nous et avec notre encadrant.

6 Support MPI / OpenMP

6.1 Notion de parallélisme

L'idée de parallélisme est née pour résoudre un problème long et coûteux en temps de calcul. Le parallélisme dans le domaine de calcul haute performance consiste à exécuter des codes en parallèle pour pouvoir augmenter la puissance des processeurs. Le parallélisme existe déjà dans les processeurs (pipeline, traitement de plusieurs instructions, ...). Le parallélisme sert aussi à multiplier les unités de traitement c'est à dire augmenter les nombres de cœurs et de dupliquer les unités vectorielles.

6.2 Notions indispensables pour le parallélisme

6.2.1 Système à mémoire partagée

C'est un système qui met en jeu plusieurs ressources de calcul. D'une manière générale, il existe deux types de système à mémoire partagée.

- a SMP ou Symmetrical Multi-Processing : C'est une machine constituée de plusieurs processeurs identiques connectés à une unique mémoire physique.
- Le NUMA ou Non-Uniform Memory Access : C'est une machine constituée de plusieurs processeurs connectés à plusieurs mémoires distinctes.

6.2.2 Système à mémoire distribuée

On dit qu'un système est à mémoire distribuée si la mémoire est répartie sur plusieurs cœurs. Les ressources de calcul n'ont pas de mémoire partagée, que ce soit de manière physique ou logicielle.

6.2.3 Thread ou flot d'exécution

C'est une implémentation de travail à faire : suite logique séquentielle d'actions résultant de l'exécution d'un programme.

6.2.4 Processus

Instance d'un programme. Un processus est constitué d'un ou plusieurs threads qui partagent un espace d'adressage commun.

6.2.5 Calcul parallèle

Le calcul parallèle consiste en le découpage d'un programme en plusieurs tâches qui peuvent être exécutées en même temps dans le but d'améliorer le temps global d'exécution du programme.

6.3 Présentation d'Open MPI

Open MPI est un outil indispensable dans le domaine de calcul haute performance. Cet outil permet de réaliser des opérations parallèles par l'interface de passage de message (Message Passing Interface). L'Open MPI est un fruit de travail de collaboration de recherche académique en partenariat avec des industries. L'Open MPI est un logiciel open source.

6.4 Installation d'open MPI

Pour installer l'outil open MPI, on a besoin de récupérer une source de l'outil dans le site officiel de Open MPI. Ensuite on décompresse la source, dans notre cas on a utilisé la version openmpi4.1.0. Pour continuer l'installation, on doit se placer dans le dossier source d'open mpi.

6.5 Configuration

Cette étape permet de configurer les différents compilateurs installés sur la machine et de définir le chemin de l'installation d'Open MPI.

6.6 Compilation d'open mpi

Pour pouvoir installer open MPI sur une machine, on doit compiler le programme dans le fichier source.

6.7 Installation

L'installation du programme se fait aussi à partir du fichier source en exécutant la commande suivante :

- `sudo make install`

6.8 Préparation environnement

Pour compiler un programme avec MPI, il faut exporter les bibliothèques nécessaires et les variables d'environnement.

- `export MPI_PATH=chemin/bin=`
- `export PATH=$MPI_PATH:$PATH`

6.9 Description de communication dans Open MPI

Comme son nom l'indique la communication dans Open MPI consiste par envoi de message. La bibliothèque MPI permet de gérer :

- l'environnement d'exécution
- les communications point à point
- les communications collectives

- les groupes de processus
- les topologies de processus

6.10 Compilation d'un programme parallèle avec **verificarlo**

Pour compiler des programmes qui font appel à la bibliothèque MPI avec le compilateur **verificarlo**, on appelle le compilateur à partir du **makefile** en ajoutant le flag suivant:

- `CC=OMPI_CC=verificarlo mpicc`

7 Vectorisation

7.1 Introduction

Différents compilateurs existent et ont des définitions de types vectoriels différents. Étant donné que notre encadrant nous a dit que le support de **gcc** était éphémère dû à une dépendance avec **fortran** qui allait être enlevée dans le futur. Nous avons décidé de ne pas supporter les types vectoriels de **gcc**. Nous ne supporterons que les types vectoriels de **clang**. Si

vous souhaitez donc tester nos tests ou nos implémentations sur vos propres codes, merci de bien vous assurer que vous avez configuré **Verificarlo** avec **clang** pour le compilateur **c** et **c++** comme suit:

```
./configure --without-flang CC=clang CXX=clang++
```

Auquel cas cela risque de ne pas fonctionner. Vous pouvez activer **flang** si vous voulez mais nous n'avons pas testé sur des codes **fortran**.

7.2 Test

Pour les tests, nous avons décidé de suivre le fonctionnement de test que **Verificarlo** a commencé à implémenter. C'est-à-dire que nous ne ferons pas de **tests unitaires** mais nous testerons si les résultats obtenus lors de la **compilation** et de l'**exécution** sont exactes. Les **tests** sont principale-

ment écrits en **bash**, avec un code de test écrit en **c** et un code **python**

qui permet uniquement de capturer les lignes où commencent et finissent les fonctions vectorielles des backends dans l'assembleurs généré à la compilation du compilateur Verificarlo par clang. Les **tests** se trouvent dans le répertoire `tests/test_vector_instrumentation/`. Les **tests** ne tes-

tent pas les **conditions**, mais uniquement les opérations **arithmétiques** sur un exemple basique. Un vecteur contient que des **1.0** et l'autre que des **1.1**. Nous avons décidé de ne pas mettre dans ce test les cas spécifiques de tout les **backends**, mais seulement s'assurer du fonctionnement pour un cas simple des opérations arithmétiques vectorielles. Pour les cas spécifiques nous pensons qu'il serait judicieux de les rajouter dans les autres tests qui test ces cas spécifiques pour un **backend** particulier pour les types de bases comme les types **float** et **double**. Nous devons testés 3

choses:

- le bon résultat des opérations vectorielles
- l'appel aux **probes vectorielles**
- l'utilisation des jeux d'instructions vectorielles (suivant l'architecture) dans les backends

Nous testons tous les backends pour les 3 sous tests, sauf pour le backend **cancellation** ou nous testons pas le bon résultat car il y a beaucoup d'**annulation** détectés et le résultat est modifié avec du bruit.

7.2.1 Bon résultat des opérations vectorielles

Pour ce faire nous devons itérer sur tout les backends, sur toutes les précisions, sur toutes les tailles de vecteurs et sur tous les types d'opérations arithmétiques en s'assurant du bon résultat à l'aide d'un fichier généré automatiquement suivant les jeux d'instruction disponible contenant le résultat attendu que l'on comparera avec la sortie de notre programme. Ce

sous-test utilise la sortie du code c. Exemple de sortie:

```
float + 4
```

```
2.100000
2.100000
2.100000
2.100000
```

Il s'agit de la sortie attendu pour l'addition du type vectorielle **float4** qui est un vecteur de 4 flotant simple précision. (addition d'un vecteur composé de 1.0 avec un vecteur composé de 1.1).

7.2.2 Appel aux probes vectorielles

Pour ce faire nous devons récupérer les fichiers **.ll**, en compilant notre fichier **c** avec **-save-temps**, qui sont les représentations intermédiaires de notre programme de test. Un fois récupéré, il nous suffit de vérifier si

l'appel aux **probes vectorielles** sont bien effectué. Exemple d'appel des

probes vectorielles:

```
%59 = call <4 x float> @_4xfloatadd(<4 x float> %55, <4 x float> %56)
...
%65 = call <4 x float> @_4xfloatmul(<4 x float> %61, <4 x float> %62)
...
%71 = call <4 x float> @_4xfloatsub(<4 x float> %67, <4 x float> %68)
...
%77 = call <4 x float> @_4xfloatdiv(<4 x float> %73, <4 x float> %74)
```

Il s'agit de la représentation intermédiaire de notre code de test. Nous pouvons voir les différents appels aux probes vectorielles pour une vecteur de 4 flotant simple précision.

7.2.3 Utilisation des jeux d'instructions vectorielles suivant l'architecture

Pour ce dernier sous-test, nous supposons que le test s'effectue sur une machine **x86₆₄** tournant sur **Linux**. Suivant les jeux d'instructions dispon-

nible sur la machine, le test vérifie si les jeux d'instructions sont bien utilisés. De plus il faut savoir que pour les processeurs **x86₆₄**, les instructions

vectérielles pour les opérations arithmétiques se compose avec la règle suivante: **opération##vectériel##presision**. Et s'utilise avec un registre vectoriel: **xmm**, **ymm** et **zmm** respectivement pour les jeux instruction **sse**, **avx** et **avx512**.

- **##**: signifie la concaténation des chaînes de caractères
- **opération**: add, mul, sub, div
- **vectériel**: **p** pour **packed** si instructions vectorielles, **s** pour **scalar** sinon
- **précision**: **d** pour double precision (double précision), **s** pour single precision (simple précision)

Par exemple, **addps** avec un registre **xmm** est une instruction vectorisé tandis que **addss** avec un registre **xmm** ne l'est pas. A noter que si nous

avons uniquement les jeux d'instructions **sse** et **avx**, nous devrions avoir des instructions **sse** pour les types vectorielles **float2**, **float4** et **double2**. Et des instructions **avx** pour tous les autres types vecorielles. Cependant

notre test, test uniquement si ces instructions sont utilisé au moins une fois et ne compte pas exactement combien de fois elles sont utilisé ce qui rendrait le test encore plus fiable. Nous supposons donc que **clang** et **llvm** vectorisent bien toutes nos opérations. Exemples de résultat attendu pour

le type vectorielles **float4**:

```
float4
2c24:c5 f8 58 c1          vaddps %xmm1,%xmm0,%xmm0
2c43:c4 c1 78 58 07        vaddps (%r15),%xmm0,%xmm0
Instruction addps and register xmm INSTRUMENTED
3024:c5 f8 59 c1          vmulps %xmm1,%xmm0,%xmm0
3043:c4 c1 78 59 07        vmulps (%r15),%xmm0,%xmm0
Instruction mulps and register xmm INSTRUMENTED
```

```

2e24:c5 f8 5c c1          vsubps %xmm1,%xmm0,%xmm0
2e43:c4 c1 78 5c 07       vsubps (%r15),%xmm0,%xmm0
Instruction subps and register xmm INSTRUMENTED
3224:c5 f8 5e c1          vdivps %xmm1,%xmm0,%xmm0
3243:c4 c1 78 5e 07       vdivps (%r15),%xmm0,%xmm0
Instruction divps and register xmm INSTRUMENTED
\vspace{5mm}

```

Il s'agit de la sortie de notre test qui affiche des bouts de code de l'assembleur du backend **ieee**. Et nous remarquons bien que les instructions vectorielles **ps** (packed single) sont bien utilisés avec les registres **xmm** qui font 128 bits.

7.3 Support des vecteurs 512 / 256 bits

Les vecteurs 512 / 256 bits était déjà supporté. Verificarlo utilise les types vectorielles de clang.

7.4 Ajout de probes vectorielles

Les probes vectorielles étaient déjà implémentés mais appelaient les probes scalaires. Nous avons donc dû modifier les probes en appelant les fonc-

tions vectorielles des backends. De plus nous avons factorisés la macro qui

permet de définir les probes vecorielles en **1** macro au lieu de **4** (une pour chaque taille) en passant la taille en paramètre.

7.5 Ajout des fonctions vectorielles dans l'interface

Il nous faut d'abord identifier quelle est l'interface et où la trouver. Nous avons facilement trouver où et comment la modifier. L'interface se trouve dans le fichier **src/common/inteflop.h**. Nous avons décidé de mettre la

taille en argument pour éviter de faire une fonction pour chaque tailles en plus d'une fonction pour chaque opérations et pour chaque précisions. Ce qui nous fait un total de 8 fonctions à ajouter au lieu de 32. Comme nous

passons la taille en argument, il faudra tester la taille pour permettre à

clang d'effectuer une opération vectorielle en changeant le type de notre tableau dans le bon type vectorielles de clang. Par exemple si nous avons

une opération flottante avec une précision **double**, l'opération **add** et un taille de vecteur de **4** nous devrons faire l'opération suivante:

```
(*(double4 *)c) = (*(double4 *)a) + (*(double4 *)b);
```

En ce qui concerne le type des opérandes, nous avons décidé de changer le type vectorielles en son pointeur sur sa précision. Reprenons l'exemple ci-dessus, pour un type **double4** nous casterons son pointeur en un pointeur de **double**. Règle: `@precision##size -> @precision` Nous pouvons faire

cela car lors de la définitions des types vectorielles, il est précisé qu'un type **precision##size** est de type **precision**. De plus nous avons dé-

placés la définitions des types vectorielles dans le fichier **src/common/inteflop.h**. Car nous avons besoins de ces types dans les **wrappers** et les **backends**. Et comme ils ont besoin tout les deux de l'interface et que ce fichier est déjà inclu dans les **wrappers** et les **backends**, il nous a paru judicieux de les déplacer ici.

7.5.1 Backend ieee

Pour le backend **ieee**, nous avons mis les opérandes constantes pour s'assurer dès la compilation que les valeurs des opérandes ne sont pas modifiées comme pour les fonctions scalaires du backends. Cependant, nous avons un **avertissement** de **clang** qui nous disait que les types des paramètres ne correspondait pas avec l'interface car nous les avons caster (changer le type) en constantes. Nous avons donc décidé d'ajouter un **pragma** qui permet de ne pas afficher l'**avertissement**. Car cet **avertissement** ne change pas le comportement de nos fonctions.

7.6 Fonctions vectorielles en mode scalaire dans les backends

Pour les fonctions **vectorielles** en mode scalaire, il suffit de prendre le code des fonctions **scalaires** et de faire un boucle sur chaque élément du

tableau. Ceci est applicable pour tout les **backends**. Nous avons implémenté les fonctions vectorielles en mode scalaire pour tout les **backends**.

7.7 Fonctions vectorielles en mode vectoriel dans les backends

7.7.1 Backend ieee

Pour le **backend ieee**, il n'y pas de traitement particulier sur les opérations. Le **backend** effectue l'opération et la débogue. Pour vectoriser l'opération,

comme dit précédemment il faut changer le type du pointeur de sa **precision** flottante en son type vectorielles de clang. Pour cela nous avons créés une macro **c** qui nous le permet. Le seul désavantage est que l'on effectue un branchement à cause de la condition. Pour la fonction de débogage,

elle est essentiellement composé de sortie standart ou dans un fichier ce qui n'est pas vectorisable. Donc nous avons laisser la boucle qui appelle la fonction de débogue pour chaque élément du tableau.

7.7.2 Backend vprec

Pour le **backend vprec**, nous avons commencé à le vectorisé. Pour l'instant il n'y a que les opérations qui sont vectorisé comme pour le **backend ieee**. Ce **backend** permet de gérer les nombre **dénormaux** (c'est-à-dire les

nombres qui ont un exposant nul). Voici un schéma qui montre la représentation d'un nombre flottant simple précision (Voir la figure 4):

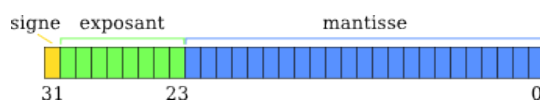


Figure 4: Représentation d'un nombre flottant simple précision

source: https://fr.wikipedia.org/wiki/IEEE_754

Revenons à notre cas, le **backend vprec** fait différentes opérations suivant si le nombre flottant est fini, infini, dénormal ou encore normal. Nous

avons commencé à réfléchir sur comment gérer les comparaisons et essayé de faire un prototype mais il n'est pas vraiment abouti et ne l'avons pas pousser sur le dépôt. Si vous êtes intéressé, le prototype se trouve ici dans

le dernier **commit**: vectorisation de vprec. Il faudra créer des structures

binary32 et **binary64** pour les types vectorielles avec des macros ce que nous avons réussi à faire. Ensuite pour toutes les opérations citées ci-dessus

il faudra tester:

1. si tous les éléments satisfont la condition
2. si il y en a au moins un qui satisfait la condition
3. si il n'y en a aucun.

Après avoir testé la condition il faudra faire les opérations. Pour le cas où tous les éléments satisfont ou non la condition, nous pouvons vectoriser les opérations. Pour le cas où il y a au moins un (et pas tous) qui satisfait la condition il faudra faire les opérations en sérialisé car le comportement ne sera pas le même pour tous les éléments du vecteur. De plus pour tester

si il y a au moins un élément du vecteur qui satisfait la condition, il faudra le tester en dernier, car nous ne voyons pas d'autres moyens que de tester séparément tous les éléments du tableau pour le moment.

7.7.3 Backend mca

Nous n'avons pas eu le temps de vectoriser le **backend mca**.

7.8 Vérification si au moins un backend utilisé implémente les opérations vectorielles

Pour l'instant seuls les backends **ieee**, **vprec** et **mca** ont été modifiés et implémentent les opérations vectorielles de façon scalaire ou vectorielle. Pour les autres backends, la version scalaire n'est même pas implémentée.

Comme pour les opérations scalaires, nous avons ajoutés dans la fonctions

d’initialisations des **probes** le fait de vérifier si au moins un **backend** utilisé implémente les opérations vectorielles. Ceci bloque tout les back-

ends qui ne les implémentent pas. Mais une sérialisation peut très vite être faites.

7.9 Compilation

Etant donné que la vectorisation implique d’utilisé les jeux d’instrucions vectorielles il faut s’assurer que les fichiers qui doivent supporter la vectorisation sont compiler avec les drapeaux des jeux d’instrucions disponibles sur la machine. Nous avons donc décidé d’utilisé le drapeaux: **-march=native**,

qui nous permet de mettre automatiquement les drapeaux des jeux d’instructions disponibles sur la machines. Nous l’avons rajouté pour la compilation des

wrappers et des **backends**. Nous avons aussi décidé de ne pas mettre

une règle pour activer ou non le drapeau, comme pour le drapeau **-Wall** dans le fichier de configuration de **autoconf**, car il nous le faut absolument pour pouvoir activer le support des opérations vectorielles, sinon il utilise uniquement le jeu d’instruction **sse**.

7.10 Problèmes rencontrés

Nous avons rencontrés plusieurs problèmes. La plupart ont pu être résolu mais il en reste un où nous n’avons pas réussis à corrigé. Il s’agit de l’optimisation de la vectorisation que permet **llvm**. C’est-à-dire que si

on compile un programme avec **clang** et que nous avons **uniquement** le jeu d’instruction vectorielle **sse** et que nous utilisons des vecteurs qui normalement représente des vecteurs **avx512** comme par exemple un vecteur de 8 éléments double précision, clang reconnait que nous n’avons pas **avx** et utilise 4 instructions **sse** à la place.

```
12e0:      66 0f 58 c4          addpd  %xmm4,%xmm0
12e4:      66 0f 58 cd          addpd  %xmm5,%xmm1
```

```

12e8:      66 0f 58 d6      addpd  %xmm6,%xmm2
12ec:      66 0f 58 df      addpd  %xmm7,%xmm3

```

Ceci est le code que clang a généré pour notre vecteur **double8** sur une machine qui n'a que **sse**. Le problème étant que aujourd'hui, **Verificarlo**

ne détecte pas 4 opérations mais qu'une seule. Nous avons donc 2 hypothèses:

- soit la phase de la détection de jeu d'instruction et de réarrangement des opérations s'effectue dans la phase d'optimisation du compilateur (ce que nous avons vu plus tôt), et donc il appelle tout de même les probes vectorielles pour des vecteurs **avx512**
- soit c'est un problème de **llvm** du fait que comme ce sont des modules différent et compilé séparément, il ne fait pas d'optimisation mais passe le vecteur par registre et donc cast (change le type) du vecteur

7.11 Connaissances acquises

Durant notre projet, nous avons acquis quelques bases sur différent logiciels ou librairies.

7.11.1 gdb

Tout d'abord avec l'aide de notre encadrant nous avons réussi à comprendre et exécuter un programme dans gdb, qui est un outils de débogue. Nous l'avons utilisé pour comprendre pourquoi **Verificarlo** ne voulais pas utilisé des instructions **sse** pour des opérations sur des vecteurs de 8 flottants simple précision par exemple, qui est un vecteur normalement utilisé avec le jeu d'instruction **avx** avec un **addpd** sur des registres **ymm** par exemple.

7.11.2 llvm

De plus durant notre exploration de **Verificarlo**, nous somme tombé sur des codes écrient en **c++** utilisant les librairies de **llvm** pour pouvoir capturer les opérations flottantes et changer ces opérations en appelant les **probes**. Nous avons compris le principe du code ainsi que la partis déjà

implémenté qui permet d'appeler les probes vectorielles. En effet un bout de codes permet rajouter dans le nom de la fonction à appeler, la taille du vecteur suivis d'un "x" pour signifié "fois". Cela se fait en testant si le type est un type vectoriel et ensuite de récupérer la taille si c'est le cas et de vérifier si c'est une taille valide de vecteur. Voici le bout de code en

question:

```
// Should we add a vector prefix?
unsigned size = 1;
if (opType->isVectorTy()) {
    VectorType *t = static_cast<VectorType *>(opType);
    baseType = t->getElementTy();
    size = t->getNumElements();

    if (size == 2) {
        vectorName = "2x";
    } else if (size == 4) {
        vectorName = "4x";
    } else if (size == 8) {
        vectorName = "8x";
    } else if (size == 16) {
        vectorName = "16x";
    } else {
        errs() << "Unsupported vector size: " << size << "\n";
        return nullptr;
    }
}
```

7.12 Conclusion vectorisation

La vectorisation étant un thème que nous avons beaucoup abordé dans le cours **architecture parallèle**, cela nous a permis de bien comprendre le sujet et d'essayer de venir à bout de cette partie. Néanmoins il reste

beaucoup de choses à accomplir comme le support des conditions (branchement) que nous n'avons absolument pas abordé que ce soit au niveau des tests ainsi qu'au niveau du support dans **Verificarlo**. Récapitulatif de ce

que nous avons fait:

- rendre les **probes vectorielles**
- implémentation des fonctions vectorielles dans les backends en mode scalaire (bitmask, cancellation, mca, mca_{mpfr}, vprec)
- implémentation des fonctions vectorielles dans les backends en mode vectoriel (ieee, vprec)
- compilation avec les jeux d'instructions disponible sur la machine de test

Récapitulatif de ce qu'il nous reste à faire:

- faire les tests sur les branchements vectoriels
- faire des tests spécifiques sur les backends pour des opérations vectorielles
- vectoriser les backends manquants (en priorité la backend **mca**)
- tester la performance sur des **NAS**

7.12.1 Performances attendues

Pour les tests sur les **NAS**, cela pourrait être intéressant de tester les opérations vectorielles avec des communications MPI ou OpenMP car nous ferons moins de communications comparé à des opérations scalaire et donc les résultats devront normalement être au rendez vous. Pour ce qui est de

la comparaison entre l'implémentation des opérations vectorielles **avant** et **après** notre projet, nous devrions avoir des gains car nous faisons moins d'appelle de fonction et nous avons vectorisé les opérations arithmétiques de base dans le cas de **ieee** et **vprec**.

8 Conclusion

Le projet nous a permis d'élargir nos connaissances sur les différents thèmes que nous avons vus dans nos cours de **Master**, comme la **vectorisation** et la **parallélisation**. Ainsi que d'autres thèmes comme la **compilation** avec le projet **llvm**. De plus il nous a permis de pouvoir contribuer à un

projet existant qui a une vocation à pourvoir contribuer au domaine du **HPC** ou du moins à avoir une première approche sur le **calcul numérique** de par la spécificité de l'outil qui a pour but de détecter les éventuelles erreurs des opérations flottantes qui est un problème majeur dans le **calcul numérique**.