

IFT6232 : Compilateur natif pour noyaux computationnels

Rapport final

Paul Khuong

23 décembre 2008

1 Introduction

Le projet vise à développer un compilateur pour des codes numériques générés semi-automatiquement, avec comme architecture cible une machine x86-64. Le générateur type serait extrait de la définition d'un algorithme de style *diviser pour régner*, qui se prête relativement bien à la parallélisation. Afin simplifier la génération, le langage source ne contient que des expressions, et permet de spécifier une forme restreinte de non-déterminisme.

Les programmes générés ont tendance à contenir de très longs blocs de base aux opérations très répétitives. Les allocateurs de registres habituels sont souvent inefficaces face à de telles entrées, alors qu'on peut souvent se restreindre à des méthodes très simples pour avoir des résultats acceptables. Il peut aussi être utile de tenter de retrouver certaines boucles afin de réduire la taille du code. En général, il faudrait donc tenter de se restreindre à des algorithmes plutôt simples et exploitant le fait que l'on travaille principalement sur un seul bloc de base.

Les machines x86-64 (à haute performance) ont une architecture superscalaire, avec plusieurs unités d'exécution asymétriques, et peuvent exécuter les instructions dans le désordre. En contrepartie, les accès à la mémoire peuvent être extrêmement longs (relativement à un calcul, e.g. multiplication de flottants), lorsqu'un accès est effectué hors de la cache. Ces coûts sont de plus extrêmement difficiles à évaluer statiquement, et peuvent varier sensiblement d'une exécution à l'autre, particulièrement en présence d'exécution concurrente. Ainsi, il est complexe de construire un modèle précis et utile du coût d'exécution d'un code, et un essai empirique n'est pas nécessairement représentatif. Il semble donc intéressant de n'effectuer qu'un ordonnancement statique grossier des instructions pour permettre à l'exécution dans le désordre d'alimenter les multiples unités d'exécution en répondant dynamiquement aux temps d'accès.

L'utilité de la mémoire cache pour pallier à la lenteur des accès mémoire a déjà été mentionnée. Typiquement, afin de mieux exploiter cette ressource, des structures de données récursives (e.g. arbre quaternaire) sont préférables à des structures plus linéaires comme des tableaux. Ces structures rendent les algorithmes diviser pour régner naturels. En plus de mieux suivre la forme des structures de données, ces algorithmes sont souvent plus facilement parallélisables. Malheureusement, les structures de données récursives mènent souvent à des calculs d'adresses complexes, et les algorithmes diviser pour régner à beaucoup d'appels récursif relativement au travail effectué. Dérouler l'arbre de calcul à partir d'un niveau fixé permet d'éviter ces deux problèmes : les structures récursives sont souvent identiques sauf pour un déplacement passé en paramètre pour un niveau donné, et dérouler la récursion permet évidemment d'effectuer plus de travail par appel. On peut ensuite se concentrer sur la performance de ces calculs complètement déroulés, ce qui peut être plus simple que de travailler sur des codes récursifs arbitraires.

Il est possible de développer, à la main, deux versions de chaque programme, une version récursive pour le haut de l'arbre de calculs et une version itérative pour les feuilles. Cependant, cela nuit à la vitesse de développement et est une source additionnelle d'erreurs (en particulier lorsque les structures de données sont modifiées). S'il était possible d'obtenir des performances comparables en extrayant la version itérative (ou déroulée) depuis la définition récursive, cette dernière approche serait certainement préférable, du moins lors du développement.

2 Rétrospective sur le travail effectué

La structure du compilateur a été largement fixée dès la proposition de projet.

1. Face avant : passage d'une représentation en S-expressions à une représentation interne plus structurée, inférence et vérification de type, et passage à une représentation interne adaptées aux analyses subséquentes.
2. Optimisations de haut niveau : vivacité des variables, propagation de constantes, copies et calculs constants, détection puis élimination des expressions communes et simplifications algébriques.
3. Optimisations de contrôle : ordonnancement des `pars`, identification et réordonnancement local des séquences parallélisables, identification des boucles.
4. Face arrière : émission de code par programmation dynamique, allocation de registre avec gestion des débordements, et optimisations par «peephole».

Les simplifications algébriques, l'identification et réordonnancement de séquences localement parallélisables, l'identification des boucles, et l'émission de code par programmation dynamique avaient été marqués comme optionnels.

Au rapport 1, seule la face avant était prête, alors qu'il était prévu d'avoir une bonne partie des optimisations de haut niveau disponibles.

Pour le rapport 2, le retard a été largement rattrapé : une forme restreinte d'élimination de variables mortes et une forme agressive de propagation des assignations ont été implantées. L'élimination des variables mortes était rendue plus complexe par la présence de blocs `par` non ordonnancés. La propagation très agressive des assignations permet des optimisations algébriques plus puissantes, et d'avoir en certains cas un code plus régulier.

De plus, les objectifs originaux ont été atteints. Les blocs `par` étaient ordonnancés explicitement et éliminés, et une face arrière naïve, avec émission de code et allocation de registre avec débordement par Belady a été implantée. Par ailleurs, afin de pouvoir utilement ordonnancer les `par`, des optimisations algébriques ont été implantées, en avance sur l'horaire prévu.

La version finale est largement différente de celle proposée ; cette différence peut principalement être expliquée par l'implantation de la détection de boucles. En effet, la détection et élimination d'expressions communes ont été fortement affaiblies puisqu'elles éliminaient parfois de la régularité dans le code. De plus, l'optimisations par «peephole» n'a pas été implantée par manque de temps. Cependant, le code actuel détecte certaines formes de boucles et émet du code par programmation dynamique, en tenant compte des registres disponibles. La détection de boucle a aussi permis d'implémenter quelques optimisations de boucle de base imprévues (en particulier «loop hoisting»), et plus simplement que si l'on travaillait dès le départ sur des boucles arbitraires. L'élimination des variables mortes et la propagation de copies & constantes classiques ont aussi été implantées pour le code complètement ordonnancé.

Ainsi, bien que certains des objectifs originaux n'aient pas été atteints (optimisations par «peephole» et élimination des expressions communes), des objectifs optionnels ou imprévus les ont remplacés : simplifications algébriques, identification de boucles, optimisation de boucles, et émission de code par programmation dynamique.

Malheureusement, une bonne partie du compilateur est codé naïvement et simplement, ce qui donne des temps de compilation médiocres. En particulier, la détection de boucle et l'ordonnancement des `par` sont superquadratiques. Cela devient problématique sur des programmes de taille moyenne, alors qu'un des objectifs originaux était de mieux traiter les blocs de base énormes que les compilateurs typiques. Des approximations ou des approches différentes permettront probablement d'éviter ces problèmes.

3 Langage source

Le code source est donné sous forme de S-expressions. Quelques formes spéciales sont offertes :

```
(let1 ([var] [valeur]) [corps])
(par [expression]+)
(progn [expression]+)
(set! [var] [valeur])
(aset! [tableau] [index] [valeur])
```

`let1` introduit une variable dans la portée statique du corps. Les expressions dans une section `par` sont évaluées dans un ordre arbitraire, mais de façon atomique, alors qu'elles sont évaluées dans l'ordre donné pour `progn`. `set!` permet d'assigner une nouvelle valeur à une variable, sauf pour les variables liées à des tableaux, qui sont immuables. `aset!` permet d'assigner une nouvelle valeur à l'élément indexé dans un tableau. Dans les deux cas, le type de la nouvelle valeur doit correspondre à celle présente auparavant (notons qu'il n'y a que des tableaux homogènes d'entiers ou de flottants).

Quelques opérateurs sont offerts : `(aref [tableau] [index])` évalue en la valeur de l'élément indexé dans le tableau. `(select [condition] [alors] [sinon])` évalue en la valeur de `alors` si `condition` évalue en vrai, et `sinon` sinon. Les expressions `alors` et `sinon` doivent être complètement pures (pas d'écriture vers variables ou tableaux). `(int [float-expr])` convertit une valeur flottante vers une valeur entière, et `(float [int-expr])` inversement. Les expressions arithmétiques `*`, `+` sont aussi disponibles (les types des deux arguments doivent être identiques, `int` ou `float`). Seuls les types des arguments doivent être fournis ; les types des autres expressions sont déduits à partir de ceux-ci. Notons que seuls les motifs de génération de code pour les accès aux tableaux et variables et l'arithmétique (entière et flottante) ont été écrits. Ajouter le support pour les autres opérateurs n'est toutefois qu'un problème de temps.

Les blocs `par` donnent une grande liberté au compilateur afin d'effectuer des optimisations de haut niveau. Les expressions peuvent être réordonnés pour effectuer des accès répétés aux mêmes adresses ou suivant une progression arithmétique. Cela permet au système de mémoire matériel de mieux détecter les motifs. Ces motifs d'accès ont aussi la particularité de pouvoir mieux se calculer via des boucles que les accès typiques des méthodes diviser pour régner. Le compilateur peut donc souvent retrouver des codes très proches de la méthode itérative qu'un humain coderait à partir des opérations effectuées par l'algorithme diviser pour régner.

4 Représentations internes

Trois représentations internes sont utilisées dans le compilateur.

La première, IR1, n'est qu'une représentation de l'arbre de syntaxe sous forme de structures. Cette représentation est utilisée principalement pour la déduction et vérification de types, et la transformations vers la seconde, mieux adaptée aux analyses.

La seconde représentation interne, IR2, ressemble à un langage de niveau C ou Fortran. Il y a une division entre blocs, opérations et expressions. Les blocs représentent une séquence d'opérations (**seq**) ou un ensemble non-ordonné d'opérations (**par**). Les opérations sont du type introduction ou élimination de variables, assignation à une variable ou écriture dans un tableau, et les expressions de l'arithmétique (entière ou flottante), sélection (dans le style de l'opérateur `? :` de C), comparaisons ou lecture de variables ou tableaux. Puisque les expressions sont complètement pures, on peut facilement effectuer des optimisations de haut niveau sur celles-ci. De même, comme les effets de bords sont restreints aux opérations, il est plus pratique d'y effectuer des optimisations telles que la propagation de constantes ou copies ou l'élimination de variables mortes.

Finalement, avant d'arriver à de l'assembleur x86-64, la génération de code émet du code à 3 adresses avec un nombre non-borné de registres virtuels entiers et flottants. Cela permet en particulier d'éliminer certaines opérations de mouvement rendues inutiles par l'allocation de registre.

5 Phases de compilation

Les phases de la face avant passent des S-expressions à l'IR2 en passant par l'IR1. Après avoir transformé les S-expressions en IR1 et assigné un type à chaque expression (et confirmé que les types concordent), une dernière phase passe à l'IR2 de façon simple.

Ainsi, à partir d'une expression $(+ x (* 2 y))$, on obtient un code formé d'une multitude d'expressions atomiques :

```
#<seq-section #21
statements:
  (#<create-var #17 var: #<tmp-var #11 type: int id: 0 name: binary-y-tmp>>
   #<assign-var #18
     var: #<tmp-var #11 type: int id: 0 name: binary-y-tmp>
     value: #<*-expression #12 type: int
       x: #<read-constant #9 type: int value: 2>
       y: #<read-var #10 type: int
         var: #<arg-var #4 type: int id: 1 name: y>>>>
   #<assign-var #19
     var: #<return-var #15 type: int id: 0 name: return>
     value: #<+-expression #16
       type: int
       x: #<read-var #13 type: int
         var: #<arg-var #5 type: int id: 0 name: x>>
       y: #<read-var #14 type: int
         var: #<tmp-var #11 type: int id: 0 name: binary-y-tmp>>>>
   #<kill-var #20 var: #<tmp-var #11 type: int id: 0 name: binary-y-tmp>>>>
```

Étant donné qu’au niveau du langage source (et donc de l’IR1), les opérateurs d’assignation peuvent être librement mêlées aux autres expressions, il n’est pas toujours évident de bien séparer opérations et expressions. Cela pose un problème puisque les expressions doivent être pures afin de simplifier les analyses subséquentes. La transformation actuelle considère toute sous-expression comme contenant un effet de bord. Chaque sous-expression est donc transformée en une opération assignant le résultat d’une sous-expression atomique (et donc assurément pure) à une variable temporaire ; le parent utilisera plutôt cette variable temporaire dans la sous-expression atomique y correspondant. Une phase subséquente pourra recréer des expressions de tailles raisonnables, en tenant compte des effets de bord.

5.1 Optimisations algébriques

La première famille de phases d’optimisations est orientée vers les optimisation algébriques. Afin d’exposer des expressions intéressantes, une première phase propage presque toutes les assignations à des variables, que la valeur assignée soit une expressions complexe ou simple, et une deuxième élimine les variables qui ne sont jamais lues (sans tenir compte des flux de données ou de contrôle) afin de minimiser la charge de travail subséquente.

Cela permet de passer des multiples expressions du dernier exemple à une seule expression :

```
#<seq-section #145
statements:
  (#<assign-var #144
   var: #<return-var #128 type: int id: 0 name: return>
   value: #<+-expression #141
         type: int
         x: #<*-expression #125 type: int
           x: #<read-constant #122 type: int value: 2>
           y: #<read-var #123 type: int
             var: #<arg-var #117 type: int id: 1 name: y>>>
         y: #<read-var #126 type: int
           var: #<arg-var #118 type: int id: 0 name: x>>>>>)
```

C’est sur de telles expressions que les optimisations algébriques sont effectuées. Plusieurs règles de réécritures sont appliquées jusqu’à l’atteinte d’un point fixe. Bien entendu, les calculs constants sont pliés autant que possible. Cependant, le traitement le plus important pour les phases subséquentes est la canonicalisation des expressions arithmétiques. Afin de pouvoir plus aisément comparer les indices, les expressions arithmétiques entières sont transformées en somme de produits via la distributivité. De plus, les additions et multiplications chaînées sont mise en un ordre canonique (arbitraire) afin d’assurer que

des calculs semblables aient des formes semblables, ce qui aide la détection de boucles.

Par exemple, l'expression $(+ (+ 3 (* 2 x)) (+ (* 2 y) (* 2 z)))$ devient :

```
#<+-expression #149
  type: int
  x: #<read-constant #23 type: int value: 3>
  y: #<*-expression #143
    type: int
    x: #<read-constant #19 type: int value: 2>
    y: #<+-expression #142
      type: int
      x: #<read-var #20 type: int
        var: #<arg-var #3 type: int id: 0 name: x>>
      y: #<+-expression #124
        type: int
        x: #<read-var #27 type: int
          var: #<arg-var #6 type: int id: 1 name: y>>
        y: #<read-var #30 type: int
          var: #<arg-var #7 type: int id: 2 name: z>>>>>>
```

Notons que le nombre de multiplications a été diminué par distributivité. Le passage vers une somme de produit permettra aussi d'identifier et comparer des expressions presque identiques (e.g. $(+ 1 (* 2 \text{stride}))$ et $(+ 2 (* 3 \text{stride}))$). Par ailleurs, même si les constantes 3 ou 2 étaient remplacées par d'autres valeurs, ces dernières occuperaient les mêmes endroits dans l'expression. Les positions des variables seraient elles aussi les mêmes. Ainsi, en plus d'effectuer des simplifications algébriques, les expressions sont réécrites afin d'en permettre une comparaison approximative, et pour amener certaines familles d'expressions semblables vers une forme commune.

5.2 Ordonnement des blocs par

La phase d'ordonnement des blocs `par` vise à imposer un ordre aux éléments de ces blocs afin d'obtenir un seul bloc de base. Puisqu'il n'y a aucune garantie sur l'ordonnement, n'importe quel ordre peut être utilisé. Cependant, il est plus intéressant d'imposer un «bon» ordonnancement ; reste à définir ce qui constitue un tel ordonnancement.

Un des buts du compilateurs est d'extraire des accès mémoires en flux. On cherchera donc à ordonner les éléments d'un `par` suivant un ordre croissant des accès mémoire. Pour cela, l'ensemble des accès à des tableaux effectués dans le `par` sont triés. Ensuite, tous les éléments effectuant le premier accès sont retirés du `par` et inséré dans un nouveau bloc `par`, puis tous ceux effectuant le second (mais non le premier), etc. Afin d'assurer une descente, un accès effectué par

tous les éléments restants est sauté. Si tous les accès sont sautés, les éléments du `par` sont placés dans un ordre arbitraire (mais encore une fois, commun avec des ensembles d'accès similaires).

Les accès sont triés topologiquement selon un ordre partiel simple. Les accès à la fois en écriture et en lectures ont priorité, puis ceux en écriture et finalement ceux en lecture. Ensuite, seuls les accès à un même tableau sont comparables, via les indices.

Chaque index de la forme $k + a_1 \cdot v_1 + a_2 \cdot v_2 \dots + x$ (où k et les a_i sont des constantes) est séparé en trois parties. La première partie est la partie variable (x). Si deux indices n'ont pas la même partie variable, ils sont incomparables. La seconde partie est la somme de $a_i \cdot v_i$. Deux indices ne sont comparables que s'ils ont exactement les mêmes v_i . S'il y a une dominance ou égalité parfaite sur les a_i , les indices sont comparables. En cas d'égalité, k est utilisé pour départager.

En général, donner priorité aux écritures et lectures combinées permet de remplacer les accès répétés au même élément d'un tableau en un accès à une variable. De plus, l'écriture en mémoire étant plus coûteuses que la lecture, il semble préférable d'assurer que les écritures répétées se fassent majoritairement vers la cache.

En particulier, pour une multiplication matricielle sur un tableau, cela amènera exactement l'ordre de parcours effectué par la triple boucle classique. Pour un ensemble d'opérations de la forme

```
(aset! dst (+ i (* stride j))
  (+ (aref dst (+ i (* stride j)))
    (* (aref x (+ i (* stride k)))
      (aref y (+ k (* stride j))))))
```

où les i , j et k sont constants et `stride` un argument, tous les indices sont comparables. De plus, l'ordre de comparaison d'index suit exactement l'ordre linéaire des adresses. Comme les écritures et lectures au même endroit ont priorité, les accès sont premièrement ordonnés par rapport à leurs accès à `dst`. On aura donc un ordonnancement équivalent à (avec n constant) :

```
for (j = 0..n) {
  for (i = 0..n) {
    [section par, pour k=0..n]
    dst[j*stride+i] += x[i+stride*k] * y[k+stride*j]
    [fin de la section]
  }
}
```

Pour l'ordonnancement des `par` restant, l'algorithme ne garanti pas d'ordre. Il est donc possible que les opérations soient ensuite triées partiellement selon leurs accès à `x` et partiellement selon ceux à `y`, ce qui pénaliserait la lecture à `x`

et y , au lieu de n'en pénaliser qu'un des deux. Imposer un ordre sur les tableaux pallierait à ce problème, mais l'utilité générale et les conséquences d'un tel choix sont difficiles à évaluer à cette étape.

Cela donne une performance très stable, quel que soit l'ordre dans lequel les opérations sont spécifiées dans la source. Un générateur basé sur une multiplication matricielle divisée pour régner, mais où la structure de matrice contient des matrices de taille moyenne aux feuilles serait un exemple de générateur où les opérations ne suivraient pas l'ordre ci-haut. Contrairement au C, pour lequel l'ordre des opérations pour une boucle de multiplication matricielle complètement déroulée peut en rendre l'exécution plus de 33% plus lente, le compilateur réordonne les opérations de façon toujours semblable.

5.3 Traitement du bloc de base

Ce n'est qu'une fois tous les blocs par éliminés (ordonnés) que l'IR2 représente un vrai bloc de base. Il est alors possible d'y effectuer la propagation de constantes et copies et l'élimination de variables mortes de façons classiques.

La propagation de constantes et copies est nettement plus conservatrice que la propagation d'assignations effectuée auparavant. Cette dernière a tendance à tout ramener en une seule énorme expression. Lorsque c'est à l'intérieur d'un petit fragment de code, avoir une seule expression aide les analyses. Cependant, il est beaucoup plus difficile d'extraire une boucle d'une expression que d'une séquence d'opérations.

L'élimination de variables mortes est elle beaucoup plus agressive. Puisque l'analyse arrière n'a à considérer qu'un vrai bloc de base, toute l'information de flux de contrôle est exposée et le flux lui-même est trivial. L'information supplémentaire permet d'éliminer plus de calculs inutiles que ce qui a été fait après le passage vers IR2 depuis IR1.

5.4 Ré-enroulage de boucles

Lorsque le compilateur sera en mesure de travailler sur des entrées de taille plus grande, l'espace occupé par le code machine exécuté pourra devenir un problème. En effet, en x86-64, encoder l'addition de deux éléments d'un tableau de flottants (doubles) consomme approximativement autant d'espace que les données (les deux flottants). Si le code machine n'entre pas en cache L1, charger ce code en cache compétitionne pour les mêmes ressources que le chargement des données (les caches de niveaux supérieurs sont souvent partagées, de même que la mémoire principale). Il devient alors avantageux de ré-enrouler le code, d'y retrouver des boucles, et ainsi réduire la taille du code machine.

Cette phase est actuellement toujours activée, bien que le compilateur ne puisse pratiquement traiter que des programmes trop petits pour dépasser une L1 typique.

L'algorithme de recherche de boucles recherche en premier des sous-séquences d'opérations similaires, et ensuite exprime les éléments différents par des progressions arithmétiques si possible. L'existence de telles progressions est en partie assurée par la phase d'ordonnancement, qui ordonne les opérations par rapport à leurs accès.

Lors de la recherche de sous-séquence d'opérations, les opérations sont abstraites en des squelettes où toutes les valeurs constantes ont été remplacées par des variables. Des opérations identiques, excepté pour les constantes, auront des squelettes semblables (idéalement identiques), tel que garanti par les réécritures algébriques. Il suffit ensuite de trouver des sous-séquences répétées dans la séquence de squelettes. L'algorithme actuel est extrêmement simple et itère à travers les longueurs de sous-séquences (bornées de 1 jusqu'à 16 opérations abstraites afin de garder des temps de compilation raisonnables) jusqu'à ce qu'une répétitions de sous-séquences soit trouvée.

Puisque la structure générale de la boucle est connue, on peut à ce moment appliquer des transformations supplémentaires sans affecter le ré-enroulement. En particulier, les expressions communes à travers les opérations similaires (entre les itérations) sont amenées hors de la boucle («loop hoisting»), mais non celles dans le corps de boucle. De plus, si des accès en écriture sont effectués répétitivement au même élément d'un tableau, tous les accès à cet élément sont remplacés par des accès à une variable (sauf s'il est impossible de déterminer que tous les accès non-remplacés ne touchent pas à cet élément). Le contenu de la variable et du tableau sont échangés à l'entrée et à la sortie de la boucle.

Lorsqu'une sous-séquence d'opérations similaires est trouvée, il faut encore pouvoir exprimer les différences en terme de la variable d'itération. Afin de garantir un code simple, le compilateur ne recherche que des progressions arithmétiques (donc formule linéaire en termes de la variable d'itération). Encore une fois, l'algorithme de recherche itère simplement à travers les longueurs de sous-séquences jusqu'à ce qu'un ensemble de progressions couvre toutes les constantes. Une série de boucles est alors émise, et la recherche reprise sur le nouveau code, afin de trouver d'autre boucles ou des boucles imbriquées.

Sur un programme simple, le produit scalaire de deux vecteurs, cet algorithme fonctionne tel que prévu. Cela permet d'obtenir le code assembleur suivant, comparable à ce qui serait écrit par un humain, pour $d = d + x \cdot y$ (un bug dans l'allocation des registres de retour nous force à passer les résultats par une référence en argument). Notons que les accès à (`aref dst 0`) ont été déplacés à l'extérieur de la boucle et remplacés par des accès à une variable temporaire dans la boucle (qui est allouée un registre).

```
; (test-dot)

    movsd (%rdi), %xmm0
    xorq %rax, %rax
    .align 16, 0x90
```

```

L1:
    movsd (%rdx,%rax,8), %xmm1
    movsd %xmm1, %xmm2
    mulsd (%rsi,%rax,8), %xmm2
    addsd %xmm2, %xmm0
    addq $1, %rax
    cmpq $64, %rax
    jne L1
    movsd %xmm0, (%rdi)

```

Cependant, sur une multiplication matricielle classique, la première itération de chaque boucle contient de multiples 0 qui sont éliminés de sommes ou éliminent des multiplications. Cela empêche l'algorithme actuel de retrouver une triple boucle parfaitement imbriquée. Quelques boucles simples et doubles et une boucle triple sont au lieu générées. Ces quelques boucles sont toutefois de loin préférables à un corps complètement déroulé lorsque la taille de la cache L1 est dépassée.

5.5 Génération de code

Un générateur de code par recherche de motifs sur arbre est utilisé pour transformer chaque opération en séquence de pseudo-assembleur. Cependant, au lieu de faire cette recherche des feuilles vers la racine, un algorithme récursif avec mémoïsation est utilisé, de la racine vers les feuilles. Cette structure permet aux motifs de travailler sur des enfants absents de l'arbre original (la complexité de l'algorithme est alors affectée). Cette méthode est bien adaptée aux demandes de l'architecture x86-64, qui offre des styles d'adressage particuliers. Notons par exemple la multiplication avec une valeur tirée directement d'un tableau dans l'exemple de la sous-section précédente.

De plus, l'algorithme considère les ordres d'évaluation et le nombre de registres (entiers et flottants) disponibles pour l'évaluation de chaque enfant, dans le style de Aho-Johnson. En effet, puisque toutes les expressions sont pures, on peut les évaluer dans un ordre arbitraire. Non seulement est il possible d'en évaluer une sous-expression avant une autre pour réduire la pression sur les registres (et les coûts), mais il est aussi possible d'évaluer une sous-expression à l'avance et de charger le résultat depuis la pile lorsque nécessaire. À ce point, le compilateur ne manipule que des registres virtuels, donc ces sous-expressions ne sont qu'évaluées bien à l'avance, et l'allocation de registre débordera le résultat sur la pile si nécessaire.

La table de mémoïsation a comme clé l'expression à émettre et le nombre de registres entiers et flottants disponibles. Comme le nombre de registres est constant, la complexité par rapport à la taille de l'expression n'est pas affectée. Lorsque le résultat d'une requête est inconnu, l'algorithme itère à travers les motifs afin de trouver ceux qui sont applicables. Pour chaque motif applicable, un ensemble d'enfant à évaluer est retourné. Toutes les permutations des ordres

d'évaluations sont évaluées, en tenant compte des registres nécessaires pour contenir les valeurs temporaires, de même que tous les sous-ensembles d'enfants évalués en avance. Notons qu'il peut être avantageux d'évaluer des enfants en avance, même si des registres sont disponibles et bien qu'il y ait une pénalité pour charger les valeurs préévaluées : lorsqu'une valeur est évaluée en avance, tous les registres sont disponibles pour évaluer les enfants. Finalement, la meilleure combinaison de motif et ordre d'évaluation est retenue, et retournée.

Le nombre maximal de registre utilisé est aussi calculé, afin d'éviter les calculs redondants : si la solution optimale pour 6 registres disponibles n'en utilise que 5, cette solution sera probablement aussi optimale pour 7, 8, ... registres libres. Il existe des contre-exemples. Cependant le nombre de registres disponibles (15 registres à entier, 16 à flottants) est si grand que l'algorithme pourrait autrement utiliser une grande quantité de temps supplémentaire pour un gain marginal ou nul.

Les motifs génèrent du pseudo-assembleur, proche de l'assembleur x86-64, mais en style trois adresses et avec une infinité de registres virtuels. Le style à trois adresses permet de mieux spécifier l'intention et de générer du code contenant moins de mouvements inutiles une fois les registres alloués. Séparer la génération de code et l'allocation de registre permet aussi de plus simplement gérer les débordements vers la pile.

5.6 Allocation de registre

La proposition originale mentionne la simplicité d'allouer les registres pour un bloc de base, même s'il est long. Malheureusement, après la détection de boucles, l'allocation de registre ne se fait pas sur des blocs de base. Cependant, la détection de boucle ne produit que des boucles bien imbriquées. Afin de prendre avantage de cette particularité, l'ensemble des variables utilisées à travers le corps d'une boucle (variables globales, traversant un bloc de base) est préalloué des positions. Ces variables sont toujours exactement en ces positions avant d'entrer dans une boucle, et avant le test de terminaison d'une boucle. À l'intérieur d'un bloc de base, un allocateur linéaire avec débordement par Belady est utilisé. Toutefois, aucun des exemples fournis n'exerce le débordement.

En parallèle avec l'allocation de registre, du code assembleur est émis (sous forme de liste de liste de chaînes, symboles et structures) à partir du pseudo-assembleur généré. Il suffit ensuite d'imprimer cette information pour avoir de l'assembleur x86-64.

6 Conclusion

Quelques tests ont été effectués sur un Opteron 2352 à 2.1 GHz non-chargé (charge moyenne < 0.03).

Le produit scalaire de vecteurs de 64 doubles, lorsque déroulé et passé à GCC (-O2) s'exécutait en 299 cycles (médiane de 100000), alors que le code généré sur entrée équivalente et présenté plus haut s'exécutait en 478 cycles. Notons qu'une boucle équivalente, écrite à la main et compilée via GCC s'exécutait en 327 cycles. Une partie de la différence peut donc être expliquée par la boucle au corps trop court. De plus, on peut rapidement voir quelques légers problèmes de performance dans l'assembleur généré.

Une multiplication matricielle classique de matrices 4×4 , déroulée et passée à GCC s'exécutait en 263 cycles, alors que le code généré par le compilateur sur une entrée équivalente s'exécutait en 487 cycles. Une boucle équivalente écrite à la main s'exécutait en 422 cycles. De plus, la fonction émise par GCC sur une entrée en désordre prenant plus de 33% autant de temps, alors que notre compilateur n'est pas affecté par l'ordre dans lequel les opérations sont écrites.

Finalement, sur une multiplication de matrices en arbres quaternaires 4×4 , le code déroulé et passé à GCC s'exécutait en 215 cycles, et celui généré par le compilateur 437. La performance de GCC est ici comparable à la boucle codée à la main et compilée en -O3. Cependant, il est probable que la situation soit différente sur des matrices de taille plus importantes.

Ces tests, en particulier celui du produit scalaire lorsque comparé à la performance de la boucle écrite par un humain, semble indiquer que la face arrière du compilateur est particulièrement faible. Cela rend l'évaluation du reste du compilateur plus difficile. Cependant, en regardant les résultats à la main, tout laisse à penser que les transformations de haut niveau ont leur place. On pourra séparer la performance de la face arrière du reste du compilateur en émettant du code pour, e.g. GCC. Une phase d'autovectorisation serait aussi très intéressante. L'interaction entre l'autovectorisation et la détection de boucles n'est toutefois pas claire. On peut espérer parvenir à insérer l'autovectorisation lors du processus de déroulement des boucles.

Toutefois, l'approche générale d'extraire des programmes semblables à des programmes itératifs depuis une spécification récursive semble réalisable. En particulier, il est possible de réordonner les opérations effectuées et d'y retrouver des boucles, du code purement itératif. La génération de code natif ne semble pas être, à priori, une bonne idée. Cependant, le code de haut niveau tel que produit par les phases d'optimisations est exactement le style de code sur lequel les compilateurs ont peu de difficultés. En particulier, l'autovectorisation amènerait la sortie de l'optimisateur près du code que les systèmes de calculs sur GPU, OpenCL et CUDA, supportent bien.