L3 MI — Programmation

Pierre Karpman
pierre.karpman@univ-grenoble-alpes.fr
https://membres-ljk.imag.fr/Pierre.Karpman/tea.html

2023–11–14

Retour aux bases

Unité *arithmétique* sur 1 bit :

- Constantes : 0, 1
- Opérations unaires : identité et négation (NOT)
- Opérations binaires : AND, OR, XOR, NAND, NOR...

Enchaîner de telles unités $\rightsquigarrow$ circuit $\ll$ binaire $\gg$

Si on ajoute une mémoire et du contrôle de flot, suffisant pour calculer tout ce qui est calculable $\rightsquigarrow$ CPU 1 bit (par ex. : PDP-8/S)

Pourquoi faire plus ?

Exemple élémentaire : multiplier deux entiers a et b

1. *Représenter* a et b dans un format compatible avec les opérations $\rightsquigarrow$ binaire $\rightsquigarrow$ besoin de $n_a := \lceil \log(a) \rceil$ et $n_b := \lceil \log(b) \rceil$ bits (si on souhaite pouvoir représenter tous les nombres)
2. Appliquer un algorithme de multiplication utilisant (uniquement) des opérations sur 1 bit $\rightsquigarrow$ coût (pour $n := n_a = n_b$) : $\Theta(n^2)$ pour l'algorithme naïf ; $\Theta(n^{1.58})$ pour l'algorithme de Karatsuba ; etc. (cf. cours d'Algo)

C'est : 1) pénible ; 2) coûteux

Que faire de plus ?

- Rendre les choses moins pénibles : fournir des fonctions de bibliothèque implémentant déjà les algorithmes (par ex., la bibliothèque math standard en C)
- Rendre les choses moins pénibles *et* moins coûteuses : fournir des *instructions* implémentant déjà les algorithmes **pour des tailles données**
  - Moins coûteux car on évite le surcoût des appels de fonction et de gestion des instructions (lecture, décodage, émission etc., cf. cours d'Archi) et on peut plus facilement paralléliser les calculs
  - Mais un investissement significatif (pas envisageable d'avoir une instruction pour chaque algorithme possiblement utile, pour chaque taille d'entrée possiblement pertinente)

Qu'a-t on de plus ?

Un processeur moderne « usuel » a généralement une unité arithmétique sur 64 bits : addition, multiplication, division etc. peuvent calculer sur des opérandes de 64 bits. *Mais on a aussi* :

- Les opérations binaires 1-bit, en version parallèle (sur 64 bits, et même parfois plus)
- Des opérations de manipulation de bits
- Des opérations sur des « vecteurs » de données, dont les « scalaires » sont stockés sur 64 bits (ou moins)

Instructions bit (2) : bitslicing

Quelles instructions pour le calcul bit (sur x86) ?

Arithmétique parallèle :

- NOT (en C : `~`)
- AND (en C : `&`)
- OR (en C : `|`)
- XOR (en C : `^`)
- ANDNOT

Manipulation :

- Décalage (en C : `<<` et `>>`)
- Décalage circulaire (rotation)
- Extraction (BEXTR, PEXT)
- Écriture (PDEP)
- Test/compatge avec ou sans extraction (POPCNT, TZCNT, LZCNT, BLSI, BLSMSK, BZHI)

Application des instructions bit

Ces instructions sont souvent utiles, par exemple pour :

- RAPPEL : Implémenter efficacement certaines opérations arithmétiques (par ex. avec des opérandes puissance de 2)
- RAPPEL : Ne pas gâcher d'espace pour stocker des valeurs binaires (par ex. représenter des ensembles)
- Calculer en parallèle sur des ensembles/vecteurs de bits

Un processeur moderne est (entre autres) un processeur ancien parallèle

Exemple : soit 64 bits $a_0, \ldots, a_{63}$, 64 bits $b_0, \ldots, b_{63}$, pour calculer a_i AND b_i pour tout $i \in [\![0, 63]\!]$:

- Avec un processeur 1-bit, on applique 64 fois l'instruction AND sur les opérandes a_i et b_i
- Avec un processeur 64-bit, on applique 1 fois l'instruction AND sur les opérandes a et b, où les 64 bits de a sont $a_0, \ldots, a_{63}$, et les 64 bits de b sont $b_0, \ldots, b_{63}$

$\rightsquigarrow$ gain de performance important (en supposant des fréquences d'horloge comparables) (gain ici principalement grâce à la parallélisation) !

Bitslicing

Le *bitslicing* consiste à étendre l'exemple précédent à n'importe quelle *fonction booléenne*

Fonction booléenne (à n bits)

Fonction à n bits d'entrée et 1 bit de sortie

- Une fonction booléenne peut toujours être calculée avec un circuit booléen/binaire (généralement non unique)
- Exemple, la fonction MAJ3 qui renvoie 1 ssi. au moins deux de ses trois entrées sont à 1 :
  MAJ3 : $a, b, c \mapsto (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$

$\rightsquigarrow$ Toute fonction booléenne peut être trivialement parallélisée en calculant son circuit sur un processeur bit parallèle

Exemple : calcul de la table de vérité de MAJ3

cf. `https://membres-ljk.imag.fr/Pierre.Karpman/maj3.c`

Bitslicing : contraintes

Contraintes d'efficacité pour le bitslicing

- On doit connaître un circuit « simple » pour la fonction à calculer (ex. : MAJ3 ; contre-ex. ad hoc : $\phi(a||b) := (a \bmod b) \bmod 2$)
  - Souvent intrinsèque au problème, mais on peut aussi passer du temps pour trouver de bons circuits (attention : la minimisation de circuit est un problème (très) difficile)
- Les entrées/sorties, les variables intermédiaires etc. doivent être écrites dans un format compatible, ou relativement rapidement convertible
  - Peut souvent être pris en compte / résolu à la conception

Bitslicing : applications

Le bitslicing est surtout utile dans des applications où des fonctions booléennes sont explicitement utilisées, par ex. :

- En cryptographie
- En algèbre linéaire sur de (très) petits corps finis (pas forcément du bitslicing au sens strict)
- Pour résoudre des problèmes SAT $\rightsquigarrow$ le dernier TP

Résolution exhaustive d'instances SAT

Rappel SAT (3CNF) (cf. cours d'Algo) :

- Entrée : une fonction booléenne à n variables spécifiée en format 3CNF : $\bigwedge_{i=1}^{c}(\vee_{j=1}^{3}(a_{i,j}))$, où dans chacune des c clauses, chaque $a_{i,j}$ est l'une des n variables (complémentée ou non)
- Sortie : SAT s'il existe une affectation des n variables qui fait que la formule s'évalue à $\top$; UNSAT s'il n'existe aucune telle affectation

Algorithme immédiat : recherche exhaustive sur les 2^n affectations possibles

- On s'arrête dès qu'on a trouvé une affectation satisfaisante
- Lors du parcours de l'ensemble des clauses, on passe à l'affectation suivante si on ne satisfait déjà plus la conjonction d'un sous-ensemble de clauses

Résolution exhaustive : itérations efficaces

L'énumération de toutes les 2^n valeurs possible des affectations peut se faire classiquement :

- En associant chaque variable au bit d'un compteur entier, qui prend les valeurs $0, \ldots, 2^n - 1$
  - Pour $n = 3$, on testera dans l'ordre :
    $(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1)$
- En utilisant un *code combinatoire* qui énumère toutes les valeurs en changeant une seule variable à la fois
  - $\rightsquigarrow$ plus rapide ici
  - (aussi éventuellement plus simple)
  - Pour $n = 3$ et le code (binaire) «de Gray» réfléchi, on testera dans l'ordre :
    $(0,0,0), (0,0,1), (0,1,1), (0,1,0), (1,1,0), (1,1,1), (1,0,1), (1,0,0)$

Code combinatoire avec des instructions bit

Problème : à chaque itération i, renvoyer l'indice j de l'unique variable dont la valeur doit être changée pour passer à l'affectation suivante

Une solution facile pour le code réfléchi : j est donné par la position du premier bit à 1 dans l'écriture binaire de i (la *valuation 2-adique* de i). Ceci est facilement calculable avec un processeur x86+BMI1, qui dispose d'une instruction dédiée à cette tâche : TZCNT (généralement) accessible depuis l'intrinsèque `_tzcnt_u64` (et ses variantes) $\rightsquigarrow$

```
if (++i >= (1ULL << n))
    return; // fin de l'énumération
j = _tzcnt_u64(i);
```

Code combinatoire avec des instructions bit (bis)

Esquisse de preuve (indice du changement donné par la valuation 2-adique) :

- Il existe une suite $\langle BF_n \rangle$ de $2^n - 1$ "flips" :
  $[0 \cdots 0]_n \rightsquigarrow [10 \cdots 0]_n$; $[10 \cdots 0]_n \rightsquigarrow [0 \cdots 0]_n$
  Immédiat pour $n = 1$, et $\langle BF_{n+1} \rangle$ peut se construire récursivement comme $\langle \langle BF_n \rangle, n+1, \langle BF_n \rangle \rangle$
  (Cf. cours d'Algo, TD6, exercice 4)
- La suite des valuations 2-adiques satisfait la même récurrence que $\langle BF \rangle$, et lui est donc égale
- Le code réfléchi à n bits est produit par $\langle BF_n \rangle$

Implémentation de la recherche exhaustive par bitslicing

Idée : avec des « slices » de 2^w bits, chaque application de la fonction booléenne peut se faire sur 2^w affectations différentes $\rightsquigarrow$ gain d'un facteur 2^w par rapport à une implémentation naïve *si le bitslicing peut se faire efficacement* $\rightsquigarrow$

- On utilise n slices x_i, une pour chaque variable
- Pour chaque $j \in [\![0, 2^w - 1]\!]$, l'ensemble des $j^{\text{ième}}$ bits des n variables x_i représente une affectation (différente)
- Idée : on fixe w x_i aux 2^w affectations possibles des variables correspondantes, et on teste les 2^{n-w} valeurs des $n - w$ variables restantes, où pour chaque test, chaque slice restante utilise la même valeur (0 ou 1) pour tous ses bits (mais cette valeur peut être différente d'une slice à l'autre !)

Exemple trivial

On prend $w = 2$, $n = 3$, et une unique clause $a \vee \neg b \vee c$

- On fixe les slices de b et c à `b = 0xA // 1010` et `c = 0xC // 1100`
- On itère sur les deux valeurs possibles pour a : 0 et 1 :
  - On assigne `a = 0x0` ; on calcule `a | (~b) | c` qui teste en une seule fois les 2^2 affectations avec $a = 0$ et toutes les valeurs possibles pour le couple b et c. On obtient `0x4`, ce qui indique que l'affectation $a = 0$, $b = 0$, $c = 1$ satisfait la formule ; on peut retourner SAT