HPC - Laboratoire 3

Rayane Annen

9 avril 2024

Partie 1

`crossjumping`

C'est une technique permettant de raccourcir la taille du code, le compilateur détecte des portions de code identiques et les rassemble, il peut modifier le control flow du programme pour qu'il garde le même comportement qu'initialement.

Dans ce premier exemple : nous avons une fonction prenant des entiers a et b en paramètres dans laquelle on calcule ensuite la somme $a + b$ mais sur laquelle a a été incrémenté de 100 s'il valait plus que 100 et b a été incrémenté de 200 dans les tous cas.

Le problème du code donné dans l'exemple et que l'incrémentation de b et le calcul de la somme est fait dans les deux branches du test de la valeur de a. Le compilateur va donc optimiser le code identique en le sortant de la boucle.

Le code ne sera pas nécessairement plus performant, toutefois la taille sera réduite.

`optimize-sibling-calls`

Une fonction à récursivité terminale (dont la dernière instruction à être évaluée est l'appel récursif) peut être optimisée, en effet la pile d'exécution de la fonction peut être réorganisée de sorte à ce qu'on la ré-utilise à chaque appel récursif, transformant alors la récursion en itération.

Dans l'exemple 2 d'une fonction inutilement compliquée pour calculer récursivement $a + b$. La différence est qu'on ne tente plus de sauvegarder l'état des registres contenant les variables et d'appeler la fonction récursive, on va simplement jump au début de la boucle lors du test, modifiant effectivement notre tail-recursion en itération.

On gagne en empreinte mémoire ainsi.

`version-loops-for-strides`

Lorsqu'on boucle sur un tableau à un certain indice, parfois on veut appliquer un stride sur celui-ci. Au lieu de se déplacer d'un pas, on se déplace d'un facteur donné (tous les strides $\times$ n pas).

Cela implique que pour chaque itération on doit multiplier l'indice courant par ce facteur.

Cette optimisation intervient alors pour créer une branche qui fait une version de la boucle quand le stride vaut 1, sans multiplication.

Dans l'exemple 3, c'est ce qui est démontré, c'est une fonction prenant un vecteur de 256 entiers et qui va incrémenter d'une valeur un certain nombre de valeurs du vecteur. Certaines valeurs sont évitées selon la valeur du stride qu'on passe en paramètres.

Le compilateur optimise alors l'opération en proposant une boucle alternative quand le stride valant 1, évitant la multiplication par 1 inutile.

Partie 2

Loop unrolling

Dans mon code j'ai, au début, créé un code qui bouclait sur le noyau pour calculer les convolutions, dans cet exemple, je fourni un code fonctionnel de calcul de convolution (filtre gaussien). Le but ici est de montrer le mécanisme de

loop unrolling et de loop peeling comme discuté en cours, comme nous avons un faible nombre d'itérations et qu'il est connu à la compilation (taille du noyau = 9), le compilateur est capable de dérouler la boucle et directement exécuter les instructions à la suite.

Ce mécanisme augmente nécessairement la taille du code cependant il n'y a plus de `jmp` + `cmp` qui est effectué pour vérifier la valeur de l'indice d'itération (puisqu'il n'existe plus). L'optimisation manuelle est de simplement nous-même dérouler la boucle.

Finalement, GCC indique toutefois que cette optimisation ne garantie pas un gain de performances.