HE VD
IG

Prof. Alberto Dassatti

High Performance Coding (HPC)
Semestre printemps 2022-2023
Contrôle continu 1
14-05-2022

Prénom: Basile

Nom: Cuneo

— Aucune documentation n'est permise, y compris la feuille de vos voisins
— La calculatrice n'est pas autorisée
— Aucune réclamation ne sera acceptée en cas d'utilisation du crayon
— Ne pas utiliser de couleur rouge

Question	Points	Score
1	6	5
2	6	4
3	12	11
4	10	10
5	6	6
6	6	5
7	10	7
Total:	56	48

Note: 5.3

Question 1: 6 points

Pourquoi les *bit-hacks* sont souvent bénéfiques à la vitesse d'exécution par rapport à l'implémentation standard d'un algorithme ? **Quand** peuvent-ils être appliqués ?

Parce qu'avec une simple suite d'opérations logiques, on peut éviter de nombreuses instructions plus couteuses telles que des branchements.

Il faut être certain que le bit hack fonctionne dans tous les cas de figure, ils sont souvent difficiles à comprendre et ne doivent en aucun cas modifier le comportement du programme.

- 1 data types

Question 2: 6 points

GCC et LLVM utilisent un flag *-march=*. Quand est-il judicieux de l'utiliser et quels compromis doivent être faits ?

Cela permet de spécifier une micro architecture et donc l'utilisation d'un pipeline, de cache, ...

Cela permet de décider des "conditions" selon lesquelles le compilateur pourra optimiser le code, il faut néanmoins avoir conscience qu'on ne peut pas faire n'importe quoi.

- 2

Question 3: 12 points

Pourquoi le compilateur ne peut pas optimiser cette fonction ?

```
void add_tensor(int *src, int *dest, size_t size){

  size_t i;

  for(i=0; i<size; i++){
    dest[i] += src[i];
  }

  return;
}
```

Parce qu'il n'a aucune certitude que les deux pointeurs pointent sur des adresses "clairement" séparées, il ne peut donc pas optimiser car les zones mémoires se chevauchent peut être.

Comment réécrire la fonction pour qu'elle soit optimisable ?

avec un ajout de vérification des plages

↳ ajouter un test comme : ||

```
if( (src+size <dest) && (dest +size) < src) {
  //version optimisée
}else{
  //version non optimisée
}
```

Comment réécrire la fonction pour utiliser des instructions **SIMD** 128*bit* (__m128*i*)? Du pseudo-code C largement commenté est parfaitement acceptable.

```
void add...
  __m128i tmp, res;          // memset des 2 vecteurs simd à 0
  int nb = size - (size % 4);   // sizeof()
  for(int i=0; i< size; i+=4){
    // chargement de 4 elements du tableau src dans tmp (tmp=_mm_load... (src[i])
    // addition de res et tmp dans res (res=_mm_add_ps(res, tmp)
    // meme chargement mais de dst dans res
    // store de res dans dst[i] avec _mm_store_ps
  }
  for(int i=nb; i<size; i++){
    dest[i] += src[i];
  }
  return;
}
```

Et **SIMD** 256*bit* (__m256*i*)? Du pseudo-code C largement commenté est parfaitement acceptable.

```
void add...
  __m256i tmp, res;
  int nb = size - (size % 8);
  for(int i=0; i<size; i+=8){
    // même corps que l'exercice d'avant mais avec les fonctions 256 bits pour gérer
    // 8 éléments à la fois.
  }                                  // 4 ways
  for(int i=nb; i<size; i++){
    dest[i] += src[i];
  }
  return;
}
```

Question 4: 10 points

Décrivez trois optimisations qu'un compilateur peut faire. Illustrez chaque exemple avec du code C.

- code motion : déplacer du code se trouvant à l'intérieur d'une boucle mais qui n'en dépend pas (calcul constant) à l'extérieur de la boucle p.ex.

ex:

```
void ma_fonction (int *tab, int i, int n, int size){
  for(int j=0; j<size; j++){
    tab[i*n+j] = j;
  }
}
```

⇒ devient :

```
int in = i*n;
for(int j=0; j<size; j++){
  tab[in+j] = j;
}
}
```

✓

- suppression du dead code : supprime les instructions et variables sans effet :

ex:

```
int global;
int myfunc(){
  int i=0;
  i++;
  i=5;
  global=3;
  i=0;
  return 0;
}
```

⇒

```
int global
int myfunc(){
  global=3;
  return 0;
}
```

✓

- constant folding : le compilateur va précalculer les calculs à base de constantes pour éviter de devoir les faire à l'exécution

ex:

```
int mydumbfunc(){
  return 2+3;
}
```

⇒

```
int mydumbfunc(){
  return 5;
}
```

✓

- il y a encore function inlining, loop unrolling, ...

Question 5: 6 points

Le code de la fonction *initializeF* peut être plus rapide que *initialize*. **Pourquoi ?**

```c
/* https://godbolt.org/
   gcc and LLVM do different things */

#include <string.h>

typedef struct{
  int a;
  char b;
  short c;
} MyStruct;

void initialize(MyStruct *s){
  s->a = 1;
  s->b = 2;
  s->c = 3;
}

void initializeF(MyStruct *s){
  memset(s,0,sizeof(*s));
  s->a = 1;
  s->b = 2;
  s->c = 3;
}
```

On ne peut pas vraiment savoir quel code sera le plus rapide, ça dépend surtout du compilateur. Il est possible que la fonction memset (...) permette de simplifier les modifications des variables, aussi, par exemple, le compilateur pourrait ensuite utiliser une "grande" valeur permettant de régler les 3 champs de la structure d'un coup.
À l'inverse, peut être que l'overhead de l'appel à memset rend le code plus lent.

Question 6: 6 points

Définissez la notion de *profiler*. **Comparez** les avantages et desavantages des approaches *Time-based sampling*, *Event-based sampling* et *program Instrumentation* pour profiler une application.

-Profiler c'est effectuer l'analyse d'un programme pour en sortir des statistiques afin de pouvoir identifier les zones "critiques" sur les quelles on passe le plus de temps et qu'on pourrait vouloir chercher à optimiser

- l'instrumentalisation est simple à mettre en place (p. ex gprof) ou manuellement (automatiquement)

:) permet d'obtenir de nombreuses statistiques.

- TBS se base sur un timer, qui, à intervale régulier, va permettre d'analyser le PC et la stack pour voir à quoi on en est, moins facile d'interpretation. *

- EBS se base sur un comptage d'événements hw ou sw, il faut ensuite comprendre à quoi on en est à partir delà, permet de ne pas trop avoir de soucis avec de longues opérations

-1

* se baser sur du temps n'est pas super en cas de longues (short!) opérations (p.ex. accès mémoire).

Question 7: 10 points

Dessinez le *roofline model* (ne pas oublier d'indiquer l'unité sur les axes) d'un ordinateur hypothétique (A) sans *cache* et *single issue*. Ajoutez sur le même diagramme l'ordinateur (B), dérivé de A, mais avec support pour instructions *SIMD FP 4-way*.
Placer le code suivant sur le diagramme pour deux machines (DA, DB) et **expliquez** votre raisonnement. (vous pouvez supposer que le compilateur est capable de émettre le code de manière optimale)

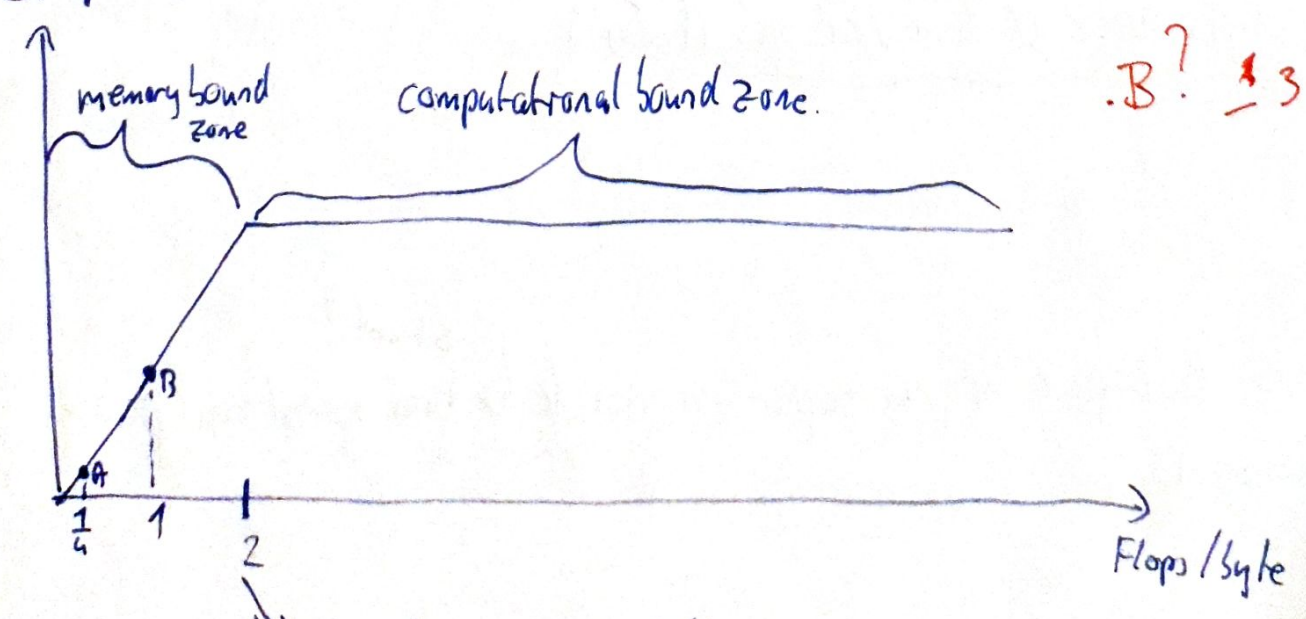
```
#define SIZE 10000000

double Ad[SIZE], Bd[SIZE], Cd[SIZE];
const int bias = 123;

void D(){
 size_t i;
 for(i=0; i<SIZE; ++i){
  Ad[i] = Bd[i] * Bd[i] + 6*Cd[i] + bias;
 }
}
```

on a 2 load de double => 2 load de 8 bytes => 16 bytes loadés
on a 4 opérations floating point

Le SIMD permet, pour un même nombre d'opérations, de traiter 4 fois plus de bytes.

=> A: on a un rapport de $\frac{4}{16} = \frac{1}{4}$

B: on a un rapport de $\frac{1}{4} \cdot 4 = 1$



B ? 3

je ne sais plus comment l'estimer...

HE IG VD | REDS Institut Reconfigurable and Embedded Digital Systems

Nom: Cuneo Basile

Intelligence Artificielle pour les Systèmes Autonòmes (IAA)

Prof. Marina Zapater (MZS)

Travail Écrit 1 (TE1)

20.04.2023

Points obtenus:	Note:
Q1 Q2 Q3 5.5 8.75 7.5	5.0

Remarques :

- Ce Travail Écrit comporte 3 parties pour un total de 30 points.
- Durée du travail écrit : **2 périodes (90 minutes)**
- Vous devez répondre sur la feuille de questions
- Vous avez le droit a :
  - Une feuille manuscrite recto-verso
  - Une calculatrice (pas utile du tout, mais vous avez le droit)
- **Aucune réclamation** ne sera acceptée en cas d'utilisation de crayon à papier
- **La couleur rouge** est réservée pour les corrections ; merci de **ne pas l'utiliser**
- Si quelque chose ne vous semble pas suffisamment précisé, veuillez décrire votre hypothèse.
- S'il y a plusieurs réponses possibles, veuillez choisir celle qui vous semble la plus simple.