OPTIMIZAÇÕES INDENPENDENTES DE HARDWARE

TeSP de Aplicações Móveis

André Martins Pereira

IPCA
INSTITUTO POLITÉCNICO
DO CÁVADO E DO AVE
ESCOLA SUPERIOR DE TECNOLOGIA

ANÁLISE DE DESEMPENHO

- Avaliar plataformas computacionais
  - PCs
  - Laptops
  - Telemóveis
  - ...
- Construir sistemas mais rápidos e eficientes

ANÁLISE DE DESEMPENHO (2)

- Métricas usadas para caracterizar o sistema
  - Latência da memória
  - Miss/hit rates
  - Velocidade de execução
  - ...
- Métricas relevantes para os factores a ser estudados
  - Melhorar a fórmula $T = N_{inst} * CPI * T_{clock}$
  - O que é que é dependente/independente do hardware?

DESMONTANDO A FÓRMULA

- T
  - tempo de execução num core do CPU
- N_{inst}
  - número de instruções executadas, dependente de
    - Compilador
    - ISA
- CPI
  - tempo médio de execução de uma instrução, depende de
    - Complexidade da instrução (adições vs divisões, etc)
    - Paralelismo na execução da instrução (a ver mais tarde)
- T_{clock}
  - período do clock, depende da microeletrónica

OPTIMIZAÇÕES

- Compiladores actuais já fazem algum tipo de otimizações
  - tentam simplificar expressões
  - usam um único cálculo de expressão em certos locais
  - reduzem cálculos repetitivos
  - usam algoritmos sofisticados para
    - alocar registos eficientemente
    - reordenação de código

OPTIMIZAÇÕES

- Compiladores actuais já fazem algum tipo de otimizações
  - limitados para garantir a execução correcta do programa
  - tem um conhecimento limitado do contexto do programa...
  - tem de compilar o código em tempo útil
- Conjunto de optimizações vedadas!

```
void f (int *x) {
   *x++;
}
```

```
int f2 (int x) {
   return 2 * f(&x);
}
```

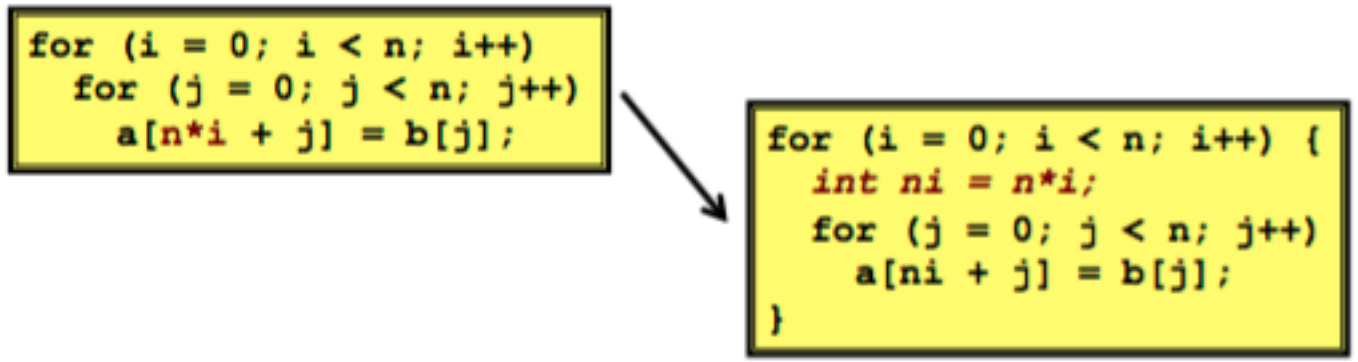
ou

```
int f2 (int x) {
   return f(&x) + f(&x);
}
```

TÉCNICAS INDEPENDENTES DA MÁQUINA

- Algumas técnicas
  - Reordenação de código para reduzir a frequência de execução
  - Simplificar cálculos substituindo expressões complexas por simples
  - Partilha de cálculos identificando expressões comuns

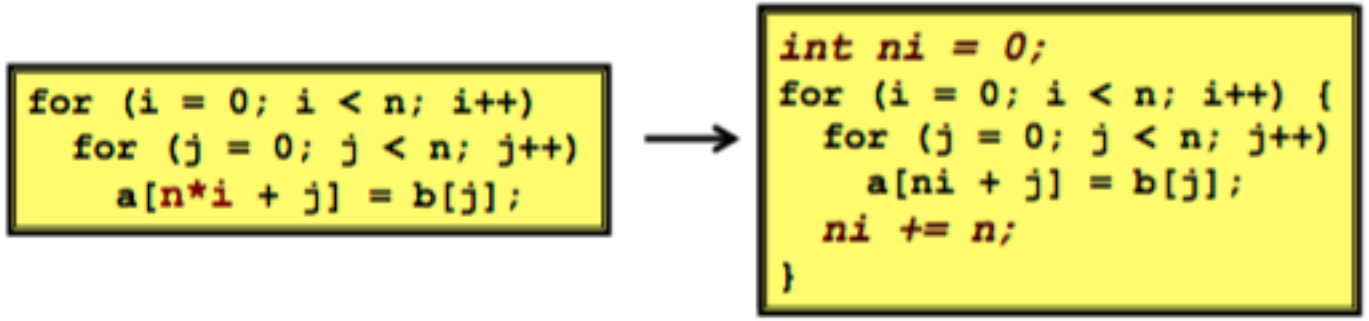
REORDENAÇÃO DE CÓDIGO



AJProença, Sistemas de Computação, UMinho, 2014/15 *10*

SIMPLIFICAR CÁLCULOS "CAROS"

- *shift* e *add* em vez de *mul* e *div*
  - 16 * x ⟶ x << 4



AJProença, Sistemas de Computação, UMinho, 2014/15 12

PARTILHAR EXPRESSÕES COMUNS

Compiladores são maus a explorar propriedades aritméticas...

```
/* Soma vizinhos de i,j */
up =    val[(i-1)*n + j];
down =  val[(i+1)*n + j];
left =  val[i*n   + j-1];
right = val[i*n   + j+1];
sum = up + down + left + right;
```

3 multiplicações: i*n, (i–1)*n, (i+1)*n

```
int inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplicação: i*n

MEDIÇÃO DE TEMPOS DE EXECUÇÃO

- O que medir na execução de um programa?
  - Por exemplo, em programas com ciclos medir o tempo por iteração
  - Cicles Per Element (CPE)
- Como medir?
  - Fazer várias medições para execuções típicas da aplicação
  - Fazer a média dos valores? Mediana?

EXEMPLO

```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

- **Procedimento**
  - calcula a soma de todos os elementos do vetor
  - guarda o resultado numa localização destino
  - estrutura e operações do vetor definidos via ADT
- **Tempo de execução** (inteiros) :
  - compilado sem qq otimização: 42.06 CPE
  - compilado com **-O2**: 31.25 CPE

AJProença, Sistemas de Computação, UMinho, 2014/15

EXEMPLO

Versão goto

```
void combine1-goto(vec_ptr v, int *dest)
{
    int i = 0;
    int val;
    *dest = 0;
    if (i >= vec_length(v))  goto done;
  loop:
    get_vec_element(v, i, &val);
    *dest += val;
    i++;
    if (i < vec_length(v))
      goto loop
  done:
}
```

1 iteração

Ineficiência óbvia:

- função **vec_length** invocada em cada iteração
- … mesmo sendo para calcular o mesmo valor!

AJProença, Sistemas de Computação, UMinho, 2014/15 20

EXEMPLO

Otimização 1:

- mover invocação de **`vec_length`** para fora do ciclo interior
  - o valor não altera de iteração para iteração
- **CPE:** de 31.25 para **20.66** (compilado com -O2)
  - `vec_length` impõe um *overhead* constante, mas significativo

```
void combine2(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

AJProença, Sistemas de Computação, UMinho, 2014/15 21

EXEMPLO

Otimização 2:

- evitar invocação de **`get_vec_element`** para ir buscar cada elemento do vetor
  - obter apontador para início do *array* antes do ciclo
  - dentro do ciclo trabalhar apenas com o apontador
- **CPE:** de 20.66 para **6.00** (compilado com **-O2**)
  - invocação de funções é dispendioso, mas tem riscos dispensá-lo
  - validação de limites de *arrays* é dispendioso

```
void combine3(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  *dest = 0;
  for (i = 0; i < length; i++) {
    *dest += data[i];
  }
}
```

AJProença, Sistemas de Computação, UMinho, 2014/15 23

EXEMPLO

Otimização 3:

- não é preciso guardar resultado em `dest` a meio dos cálculos
  - a variável local `sum` é alocada a um registo
  - poupa 2 acessos à memória por ciclo (1 leitura + 1 escrita)
- **CPE:** de 6.00 para **2.00** (compilado com **-O2**)
  - acessos à memória são dispendiosos

```
void combine4(vec_ptr v, int *dest)
{
  int i;
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int sum = 0;
  for (i = 0; i < length; i++)
    sum += data[i];
  *dest = sum;
}
```

AJProença, Sistemas de Computação, UMinho, 2014/15 24

EXERCÍCIO (1)

```
int maior(int vector[]){
  int i;
  int max = -MAX_INT;
  for(i=0;i<size(vector);
i++){
    if(vector[i] > max){
      max = vector[i];
    }
  }
  return max;
}
```

```
int find(int num, int
vector[]){
  int s = size(vector);
  int res = -1;
  for(i=0; i<s; i++){
    if(vector[i] == num)
      res=i;
  }
  return res;
}
```

```
int find_mult(int num,
int vector[]){
  int s = size(vector);
  int res = -1;
  for(i=0; i<s; i++){
    if(vector[i] <= num)
      res*=vector[i];
  }
  return res;
}
```

EXERCÍCIO (2)

```
typedef struct str{
int key;
int values[30];
} Hash;

Hash list[50];

//procura no array "values" da estrutura "Hash h" se existe o valor "num"
int existe(Hash h, int num);

int process(Hash h){
  int num = 0;
  for(i=0; i<50; i++){
    num = i+1;
    if(existe(hash[i], num) != -1){
      h[i].values[existe(list, num)] = 0;
      res += x*10;
    }
  }
  return res;
}
```

EXERCÍCIO (3)

- Implementar a função **mult_impares**, que recebe uma matriz de inteiros e devolve a multiplicação de todos os números ímpares dessa matriz
  - Exemplo: para a matriz [2,3][4,5], deve devolver 3*5=15.

```
int matrix[20][20];

int mult_impares(int matrix[][]);
```