

Parte 9

Otimização Básica Código Serial

CI1164 - Introdução à Computação Científica
Profs. Armando Delgado e Guilherme Derenievicz
Departamento de Informática - UFPR

Tipos de Armazenamento de variáveis

- **Pilha**: variáveis locais, alocadas no início de execução de funções
 - *First-in-last-out*: mesma região de memória, usadas seguidas vezes
 - ▶ Ficam em cache **L1** a menos que hajam arrays grandes
 - Sempre dê preferência a variáveis locais:
 - ▶ Elas ficam próximas umas das outras em memória e portanto em cache
- **Estática** ou **global**: parte estática da memória
 - constantes, variáveis globais ou variáveis estáticas
 - Podem ser inicializadas antes do início do programa
 - Performance de cache ruim porque elas não estão próximas às variáveis locais

Tipos de Armazenamento de variáveis

- **Register**: variáveis mantidas diretamente em registradores
 - Compiladores automaticamente colocam variáveis mais usadas em registradores.
 - Depende fortemente da quantidade de registradores disponíveis
- **Dinâmico**: Alocado com `malloc` (`new` em C++)
 - Alocação / desalocação consome tempo
 - Memória (*heap*) pode se tornar fragmentada
 - Programador deve gerenciar ponteiros

Alocação de Memória

- Sempre alocar memória contígua

→ Ex.: alocar array 2D

```
double *a;  
a = (double*) malloc( n*n*sizeof(double));  
  
double **a;  
a = (double**) malloc( n*sizeof(double*));  
for (int i=0; i<n; ++i)  
    a[i] = (double*) malloc( n*sizeof(double));
```

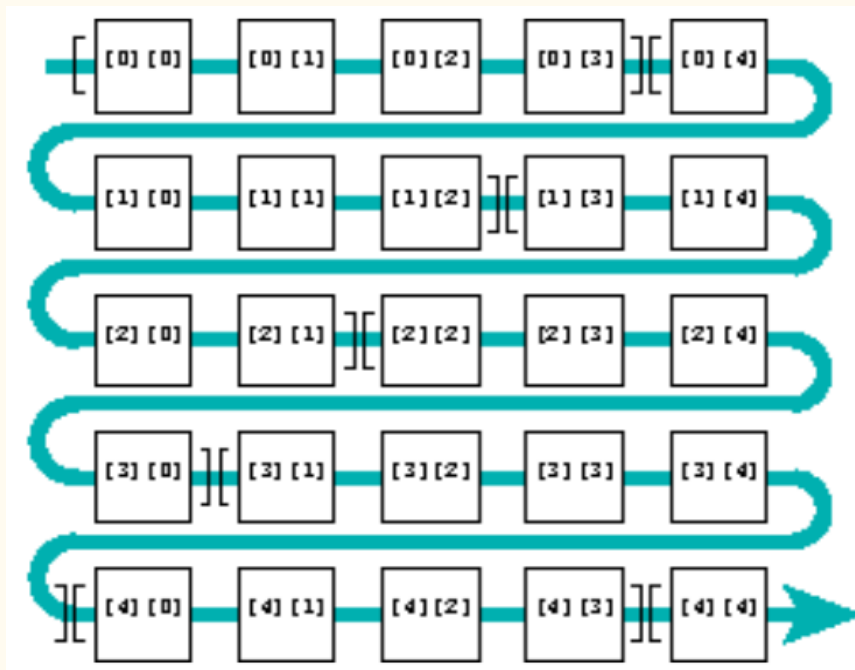
- Layout* de dados → permitir computações feitas em cache

→ Ex.: estrutura de coordenadas cartesianas

```
struct Pontos { double *x, *y; } p1; // Struct of Arrays  
p1.x = (double*) malloc(n*sizeof(double));  
p1.y = (double*) malloc(n*sizeof(double));  
  
struct Ponto { double x,y; } *p2; // Array of Structs  
p2 = (struct Ponto *) malloc(n*sizeof(struct Ponto));
```

Layout de dados e acesso contínuo

- Matrizes: layout default para **C/C++** → linha a linha (*row major order*)



```
for(i=0; i<N; ++i) {  
    for(j=0; j<N; ++j) {  
        a[i][j] = i*j;  
    }  
}
```

Acesso contínuo ! 😊

```
for(j=0; j<N; ++j) {  
    for(i=0; i<N; ++i) {  
        a[i][j] = i*j;  
    }  
}
```

Acesso com stride N ! ⚡

Arrays

- Em um laço, um array multidimensional deve ter seu último índice mudando mais rápido (laço mais interno)
 - Tamanho da dimensão associada deve ser uma potência de 2:
 - ▶ Simplifica cálculo de endereço em acesso não-sequencial
`double matrix[L*C];`
posição $(i, j) \rightarrow \text{matrix}[i*C+j]$
 $C = 2^k \rightarrow i*C+j = i \ll k + j$
 - ▶ Cuidado com **cache thrashing**
- Use `memset()` para inicializar memória
 - Cuidado com código paralelo (**write allocation**)

Eliminação de subexpressões

Ao invés de:

$$q = a + b + c$$

$$p = a + b + d$$



Compilador avalia:

$$t = a + b$$

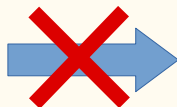
$$q = t + c$$

$$p = t + d$$

Mas compilador não substitui:

$$q = a + b + c$$

$$p = a + d + b$$



com:

$$t = a + b$$

$$q = t + c$$

$$p = t + d$$

Ao invés de:

$$q = f(x) + b * f(x)$$



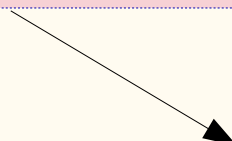
escreva:

$$q = f(x) * (1 + b)$$

Medidas simples, impacto grande

- Eliminação de subexpressões comuns

```
for (i=0; i<n; ++i)  
    A[i] += i * (s+r*sin(x));
```



```
aux = s+r*sin(x);  
for (i=0; i<n; ++i)  
    A[i] += i*aux;
```


Medidas simples, impacto grande

- Existem várias formas de evitar sobrecargas

→ Por exemplo:

substitua

```
if (sqrt(tt) < eps) { ... }
```

por

```
if (tt < eps*eps) { ... }
```

Otimizações de bom senso

- Faça menos trabalho!

```
FLAG = 0;  
for (i=0; i<n; ++i)  
    if (complex_Func(A[i]) < THRESHOLD)  
        FLAG = 1;
```



```
FLAG = 0;  
for (i=0; i<n && !FLAG; ++i)  
    if (complex_Func(A[i]) < THRESHOLD)  
        FLAG = 1;
```

Otimizações de bom senso

- Evite operações / funções que tomam tempo

```
int iL, iR, iU, iO, iS, iN; // spin: +1 (up) ou -1 (down)
double edelz, tt;

... // carrega valor 'tt'
for (...) // um laço muito longo
    ... // carrega valores de spin
    edelz = iL+iR+iU+iO+iS+iN;
    BF    = 0.5*(1.0 + tanh(edelz/tt));
```

- ➔ `tanh()` é uma função muito cara
- ➔ `edelz` tem um faixa fixa de valores ($-6 \leq \text{edelz} \leq +6$)

Otimizações de bom senso

- Utilize uma tabela de *lookup*:

```
int iL, iR, iU, iO, iS, iN; // spin: +1 (up) ou -1 (down)
double tt, tanh_table[13];

... // carrega valor 'tt'
for (int i=-6, i <= 6; i++) // Este laço cria a lookup table
    tanh_table[i+6] = 0.5*(1.0 + tanh((double) i/tt));
...
for (...) // um laço muito longo
    ... // carrega valores de spin
    BF = tanh_table[iL+iR+iU+iO+iS+iN + 6];
```

- ➔ Tabela é pequena e permanece na cache durante todo o laço do cálculo principal

Otimizações de bom senso

- Reduza seu conjunto de dados → memória acessada
 - ➔ Aumenta probabilidade de *cache hits*
- Use representação de valores em tipos “menores”
 - ▶ Para valores pequenos (exemplo anterior)
 - ▶ Atente ao alinhamento de memória
 - ▶ Inteiros de 1 byte podem não ser muito efetivos
- Não acesse dados desnecessários
- Não inicialize memória antes que seja necessário
- Não faça realocação de memória
 - ▶ `malloc` → `memcpy` → `free`

Medidas simples, impacto grande

- Evite desvios

→ Prejudicam *pipeline* e otimizações pelo compilador

```
for (i=0; i<n; ++i)
    for (j=0; j<n; ++j) {
        if (j < i)
            sign = 1.0;
        else if (j > i)
            sign = -1.0;
        else
            sign = 0;
        C[i] = C[i] + sign * A[i][j] * B[j];
    }
```

Medidas simples, impacto grande

- Evitando desvios

```
for (i=0; i<n; ++i)
    for (j=0; j<i; ++j)
        C[i] = C[i] + A[i][j] * B[j];

for (i=0; i<n; ++i)
    for (j=i+1; j<n; ++j)
        C[i] = C[i] - A[i][j] * B[j];
```

Checagem de limites

- Para verificar se um inteiro está dentro de um certo intervalo:

```
if (i >= min && i <= max) { ...
```

- Pode ser feito como:

```
if ((uint)(i - min) <= (uint)(max - min)) { ...
```

- Forma mais rápida de limitar a faixa de valores de um inteiro

- comprimento do intervalo desejado é uma potência de 2:

```
float list[16]; int i;
```

```
...
```

```
list[i & 15] += 1.0f; // i=18 → list[2]
```

- Previne erros em tempo de execução!

Medidas simples, impacto grande

- Usando instruções SIMD:
 - Use dados sequenciais em memória
 - Use ponteiros alinhados em memória
 - ▶ Endereços de memória divisíveis por 16
 - ▷ às vezes pelo tamanho da linha de cache
 - ▶ `void *aligned_alloc(alignment, size)`
 - ▶ Laço deve começar em uma posição alinhada (**-falign-loops** ou **-O3**)

Medidas simples, impacto grande

- Usando instruções SIMD:

→ Evite dependência de dados dentro de laços

```
for (int i=0; i < N; ++i)
    a[i] = s * a[i-1];
```

→ Evite desvios dentro de laços

→ Compilador pode fazer a “mágica”

- ▶ o uso de funções SIMD intrínsecas pode melhorar a performance
 - ▷ Funções com correspondência direta com instruções assembly
 - ▷ Dependem do compilador

Compiladores

- Opções gerais de otimização
 - `-O1`, `-O2`, `-O3`,
 - `-Ofast` (`-ffast_math`)
 - ▶ `-funsafe-math-optimizations ...`
 - ▶ `-fassociative-math ...`
 - Alguns problemas aparecem apenas em alta otimização
 - Defina a arquitetura
 - ▶ `-march=?`

Compiladores

- Precisão computacional
 - Por definição, regras de associatividade não se aplicam à matemática de Ponto Flutuante
 - *Denormalização* também diminui a velocidade de computação.
 - ▶ Desabilitação (*flush to zero*) em **hardware** pode ser feita
- Otimização de Registradores
 - *Inlining* ajuda a manter dados em registradores
 - ▶ Expressões aritméticas grandes podem esgotar o uso de registradores
 - Existe uma limitação de registradores (8 a 128)
 - Se há carência de registradores, variáveis devem ser expulsas, i.e., escritas em memória
 - ▶ Efeito pode ser verificado pelos contadores de hardware

Compiladores

- *Aliasing*

```
void scale_shift(double *a, double *b,  
                double s, int n) {  
    for(int i=1; i<n; ++i)  
        a[i] = s*b[i-1];  
}
```

→ Se **a** e **b** apontam para o mesmo **objeto** na memória (por, exemplo, um mesmo vetor)

- ▶ Ocorre uma “dependência real”,

- ▷ Nenhuma otimização pode ser aplicada (e.g.: não é possível usar SIMD)

→ Compilador deve ser informado que não há “alias”

- ▶ -O2 ou -O3

- ▶ Palavra chave `restrict`

- ▷ `void scale_shift(double * restrict a, double * restrict b, ...);`

Compiladores

- *Inlining*
 - Insere um código completo de uma função no ponto de chamada
 - ▶ Remove a necessidade de argumentos na pilha
 - Não se deve confiar no compilador para realizar *inlining*
 - ▶ Use a palavra-chave *inline*
 - Aumenta o código objeto (mais L1I misses)
 - Pressão sobre registradores aumenta

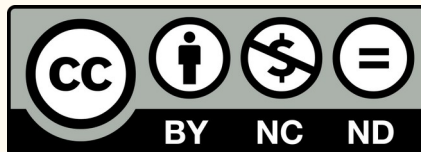
Referências

- Daniel Weingaertner; notas de aula da disciplina **Introdução à Computação Científica (UFPR/DINF)**
- G. Hager, G. Wellein; **Introduction to High Performance Computing for Scientists and Engineers**. CRC Press, 2011.

Créditos

Este documento foi desenvolvido pelo Prof. Armando Luiz N. Delgado (UFPR/DINF), para uso na disciplina Introdução à Computação Científica (CI1164), a partir de conteúdo de autoria do Prof. Daniel Weingaertner (UFPR/DINF).

Compartilhe este documento de acordo com a licença abaixo



Este documento está licenciado com uma Licença Creative Commons **Atribuição-NãoComercial-SemDerivações** 4.0 Internacional.

<https://creativecommons.org/licenses/by-nc-sa/4.0/>