Parte 10

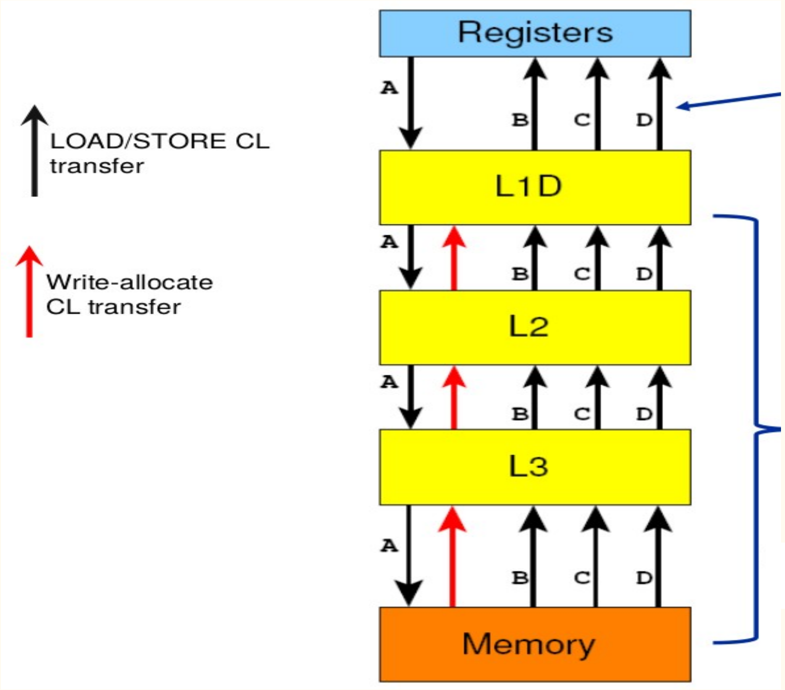
Otimização de Acesso a Dados

CI1164 - Introdução à Computação Científica

Profs. Armando Delgado e Guilherme Derenievicz
Departamento de Informática - UFPR

Análise de Transferência de Dados: tríade com vetores

- `A[i] = B[i] + C[i] * D[i]`



- Operações de Load/Store (LD/ST)
  - ➔ Tamanho do registrador (não CL)
- Transf. de dados SEMPRE em linhas de cache (CL)
- Supondo **double** e
  1 CL = 64 Bytes = 8 double's
  - ➔ 1 CL = valores para 8 iterações
  - ➔ `A[i:i+7] = B[i:i+7] + C[i:i+7] * D[i:i+7]`

Análise de Transferência de Dados: tempo de transferência

Hierarchy	Transfer time to next level	Total transfer time
L1	3 cy	**3 cy**
L2	5 cy	**8 cy**
L3	10 cy	**18 cy**
Memory	23 cy	**41 cy**

Ciclos por linha de cache no processador Intel Xeon E5-2695 v3 (“Haswell”)

Equilíbrio de código

- **Equilíbrio de código** (**B_C**): quantifica quantidade de dados transferida por unidade de trabalho

$$B_c = \frac{transferência\ de\ dados\,[Byte]}{quantidade\ de\ trabalho\,[flops, iterações, \ldots]}$$

- Exemplo: Tríade de vetores, precisão dupla (double)
  - ➔ B_C = (4+1) words / 2 Flops = 2,5 W/Flop = 20 Bytes/Flop
- Para um determinado trecho de processamento (*kernel*), determine em função do tamanho do problema
  - ➔ A **quantidade mínima de transferência de dados**
  - ➔ O **número de operações de ponto flutuante** a ser executado

Exemplo: Multiplicação Matriz-matriz

```
double a[N][N], b[N][N], c[N][N];
for (i=0; i<N; ++i)
  for (j=0; j<N; ++j)
    for (k=0; k<N; ++k)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

- Transferência de dados: 3 (NxN) matrizes → 3 * N^2 * 8 **B**ytes
- Quantidade de trabalho: 3 laços aninhados → 2 * N^3 * **F**lops
- $B_C = \frac{24 * N^2}{2 * N^3} B/F \rightarrow \frac{O(N^2)}{O(N^3)}$

Caso 0: Transposta de matriz

- Cálculo da transposta de uma matriz densa **$A = B^T$**
  - ➔ acesso em passo (***stride***) maior que 1 à memória em **A** ou **B**

```
for (int i=0; i<N; ++i)
    for (int j=0; j<N; ++j)
        A[j][i] = B[i][j];
```

versus

```
for (int i=0; i<N; ++i)
    for (int j=0; j<N; ++j)
        A[i][j] = B[j][i];
```

- Devido ao processo de **write-allocate**
  - ➔ Custo de escritas *strided* > Custo de leitura *strided*

Cache Thrashing

- Acesso *strided* também pode levar a ocorrência de ***cache thrashing*** se a dimensão do vetor é uma potência de 2

➔ Inserir um *padding* na dimensão mais inferior (*leading*) (B[N][N+p])

➔ **Regra de ouro:** fique longe de potências de 2 em dimensões inferiores de vetores

  ▷ Tente fazer estas dimensões serem múltiplos ímpares de 16

Caso 1: Algoritmos *O(N)/O(N)*

- Transferências de dados *O(N)* vs. Operações aritméticas *O(N)*
  - ➔ Exemplos: Produto escalar, soma de vetores, multiplicação matriz-vetor (MVM), etc.

- Performance limitada por memória para valores grandes de N (“*memory-bound*”)

- Potencial de otimização limitada para laços simples

Case 1: Algoritmos *O(N)/O(N)*

- Exemplo: somas sucessivas de vetores

```
for (int i=0; i<N; ++i)
    A[i] = B[i] + C[i]

for (int i=0; i<N; ++i)
    Z[i] = B[i] + E[i]
```

➔ Nenhum potencial de otimização em qualquer um dos laços

➔ Performance limitada pela banda de memória (*memory bound*)

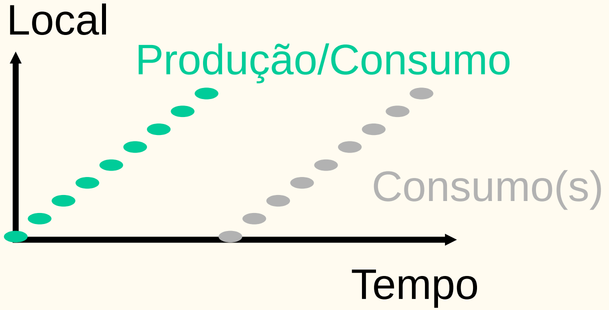
➔ Fusão de laços (***Loop fusion***)

```
for (int i=0; i<N; ++i) {
    A[i] = B[i] + C[i]
    // Economiza um LOAD para B[i]
    Z[i] = B[i] + E[i]
}
```

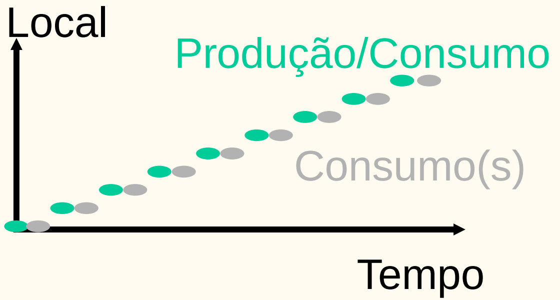
▷ permite *O(N)* reuso de dados de registradores

Fusão de Laços (*Loop Fusion / Merging*)

```
for (i=0; i<N; i++)
    B[i] = g(A[i]);
for (j=0; j<N; j++)
    C[j] = f(B[j],A[j]);
```




```
for (i=0; i<N; i++){
    B[i] = g(A[i]);
    C[i] = f(B[i],A[i]);
}
```



- Aumenta localidade
- Reduz sobrecarga do laço

Fusão de Laços (*Loop Fusion / Merging*)

```
for (i=2; i<N; i++)
   B[i] = f(A[i]);
for (i=0; i<N-2; i++)
   C[i] = g(B[i+2]);
```

i+2 > i ⇒ fusão não é possível

```
for (i=2; i<N; i++)
   B[i] = f(A[i]);
for (i=2; i<N; i++)
   C[i-2] = g(B[i+2-2]);
```

i+2–2 = i ⇒ fusão possível

Fusão dos laços ⇒

```
for (i=2; i<N; i++) {
  B[i] = f(A[i]);
  C[i-2] = g(B[i]);
}
```

Fusão de Laços (*Loop Fusion / Merging*)

- Dependências podem bloquear a fusão do laço
  - ➔ Muda-se o laço para contornar a dependência

```
for (i=0; i<N; i++){
   A[i] = f(A[i-1]);
   B[i] = g(in[i]);
}
for (j=0; j<N; j++)
   C[j] = h(B[j],A[N-1]);
```

Quebra corpo do laço

```
for (i=0; i<N; i++)
   A[i] = f(A[i-1]);
for (k=0; k<N; k++)
   B[k] = g(in[k]);
for (j=0; j<N; j++)
   C[j] = h(B[j],A[N-1]);
```

Fusão de laço

```
for (i=0; i<N; i++)
   A[i] = f(A[i-1]);
for (j=0; j<N; j++){
   B[j] = g(in[j]);
   C[j] = h(B[j],A[N-1]);
}
```

Caso 2: Algoritmos *$O(N^2)/O(N^2)$*

- Cenários típicos são laços aninhados em 2 níveis onde cada laço tem uma contagem de **N**, with $O(N^2)$ operações para $O(N^2)$ ***loads*** e ***stores***

- Exemplos: multiplicação matriz-vetor, soma de matrizes, transposição de matriz, etc.

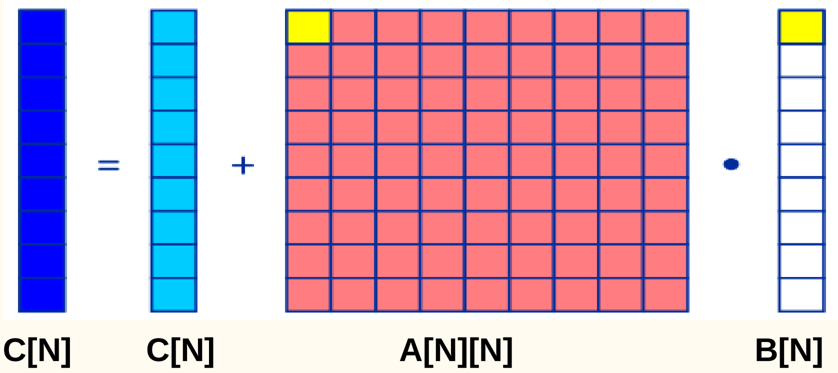
- Limitado em memória para valores grandes de **N**

- Algum potencial de otimização (fator constante)

Caso 2: Algoritmos $O(N^2)/O(N^2)$

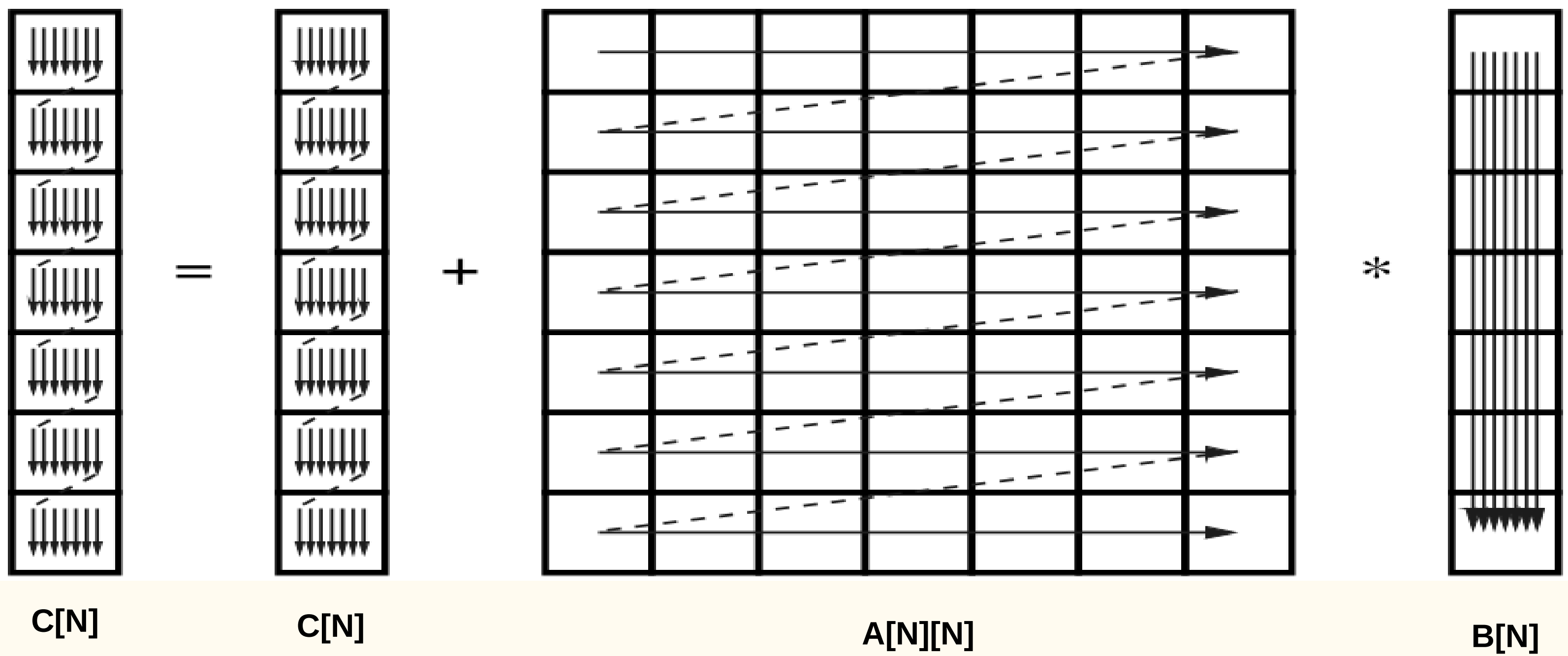
- Exemplo: Multiplicação matriz-vetor (MVM)

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



➔ Versão simples carrega B **[ ]** **N** vezes !

Padrão de Acesso a Memória (N=7)



MVM → Loop unrolling

- O laço externo é atravessado com um passo (*stride*) de **m**
  - ➔ laço interno é repetido **m** vezes;

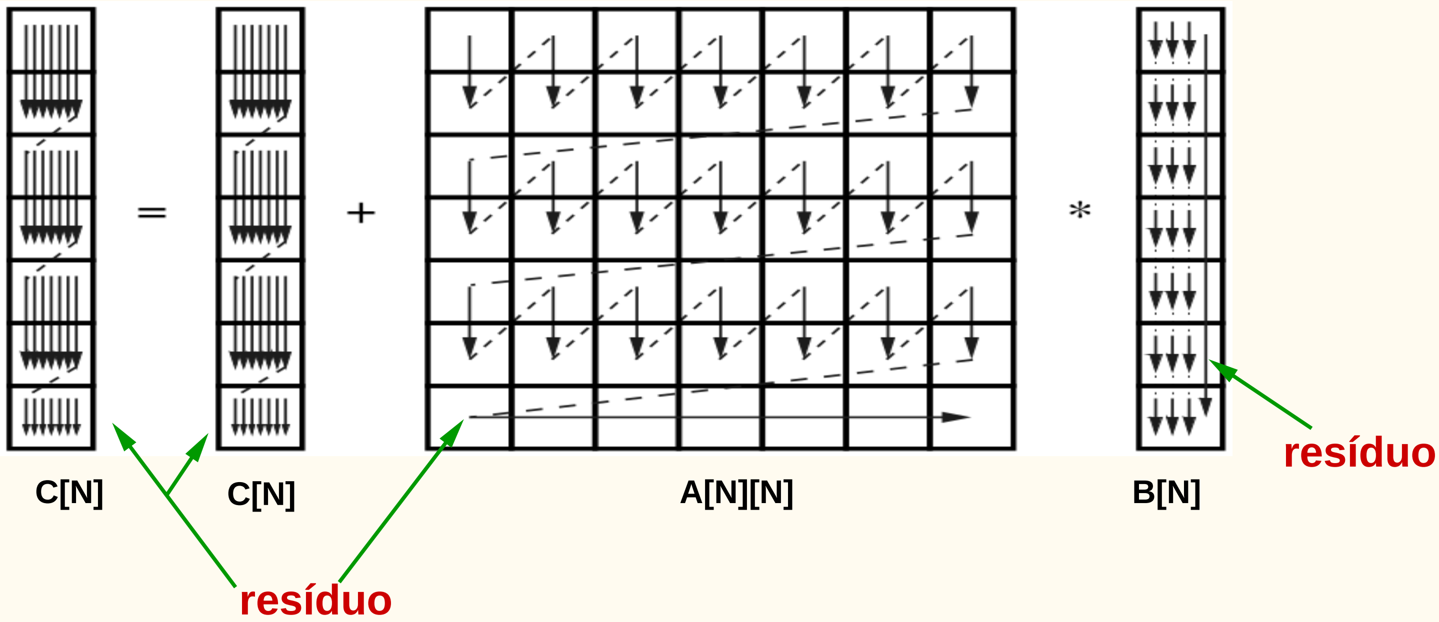
```
for (int i=0; i <N-N%m; i+=m){
    for (int j=0; j<N; ++j)
        C[i] = C[i] + A[i][j] * B[j];
    for (int j=0; j<N; ++j)
        C[i+1] = C[i+1] + A[i+1][j] * B[j];
    //...
    for (int j=0; j<N; ++j)
        C[i+m-1] = C[i+m-1] + A[i+m-1][j] * B[j];
}
// resíduo do laço("remainder loop")
for (int i=N-N%m; i < N; ++i)
     for (int j=0; j < N; ++j)
        C[i] = C[i] + A[i][j] * B[j];
```

MVM → Loop Unroll & Jam

- No exemplo anterior, um simples *unroll* não oferece ganhos em performance
- **Solução:** *Unroll* + *Fusão* → *Unroll & Jam* !

```
for (int i=0; i < N-N%m; i+=m)
    for (int j=0; j<N; ++j){
        C[i] = C[i] + A[i][j] * B[j];
        C[i+1] = C[i+1] + A[i+1][j] * B[j];
        //...
        C[i+m-1] = C[i+m-1] + A[i+m-1][j] * B[j];
    }
// resíduo do laço(“remainder loop”)
for (int i=N-N%m; i < N; ++i)
    for (int j=0; j < N; ++j)
        C[i] = C[i] + A[i][j] * B[j];
```

MVM → Loop Unroll & Jam (m = 2, N=7)



MVM → Loop Unroll & Jam

- Ganhos de performance:
  - ➔ **B** é carregado **N/m** vezes ao invés de **N** vezes
- Ciladas na performance:
  - ➔ Necessários mais registradores (um para cada linha da matriz **`A`**)
  - ➔ Mais linhas da matriz **`A`** em cache
  - ➔ Se linha de matriz muito grande
    - ▷ Mapeamento de endereços de memória prejudica performance
      - ▶ *TLB misses*
  - ➔ Risco de *cache thrashing*

Outro Exemplo: MVM com acesso por coluna

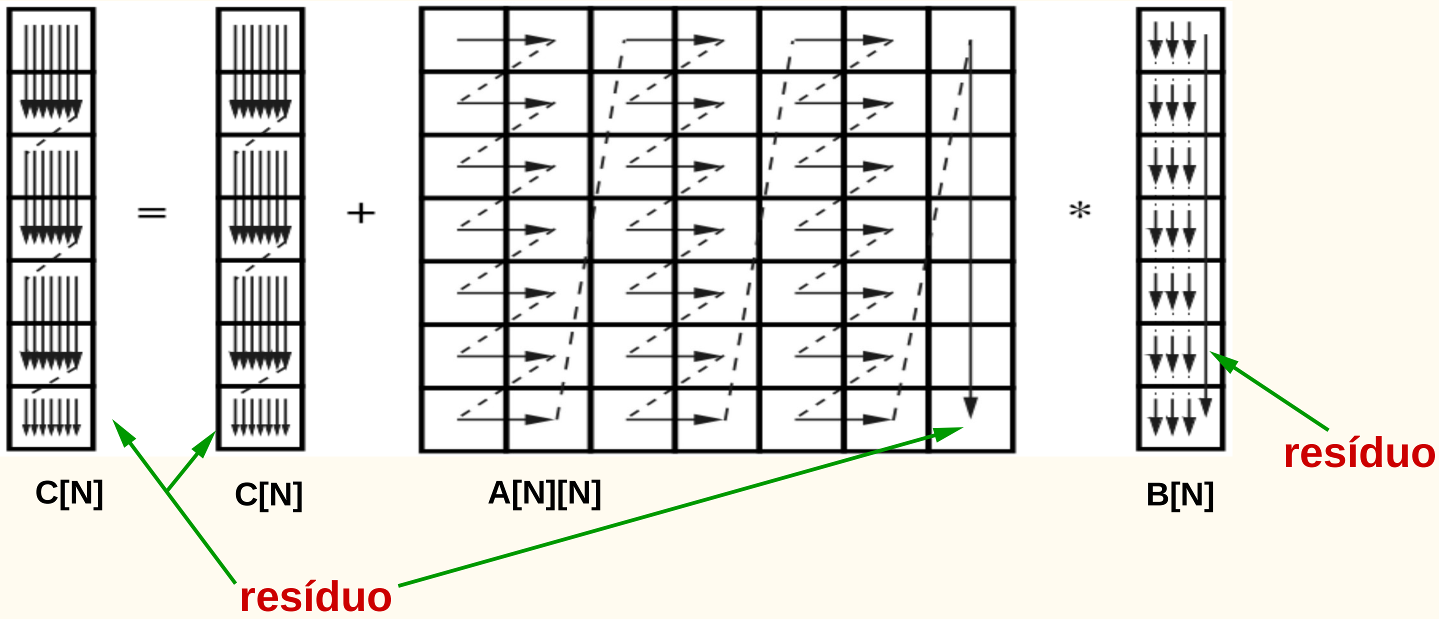
- Exemplo: multiplicação matriz-vetor, acesso à matriz **por coluna**
  - ➔ Versão simples carrega **B[ ]** **N** vezes !
  - ➔ **A** é acessado em ordem de coluna (uso ruim de cache)

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

  - ➔ ***Unroll & jam:***

```
for (int j=0; j<N-N%m; j+=m)
     for (int i=0; i<N; ++i) {
          C[j] = C[j] + A[i][j] * B[i];
          C[j+1] = C[j+1] + A[i][j+1] * B[i];
          //...
          C[j+m-1] = C[j+m-1] + A[i][j+m-1] * B[i];
   }
...
```

MVM (coluna) → Loop Unroll & Jam (m = 2, N=7)



Caso 2: Algoritmos $O(N^2)/O(N^2)$

- Exemplo: Transposição de matriz → **A = B**^{T}

```
for (int i=0; i<N; ++i)
    for (int j=0; j<N; ++j)
        A[i][j] = B[j][i];
```

- Matrizes **A** e **B** carregadas/escritas em memória exatamente 1 vez
- *Unroll & Jam* pode dar um ganho aproximado de 50% em performance

Transposta → Loop unrolling

```
for (int i=0; i<N; ++i)
    for (int j=0; j<N; ++j)
        A[i][j] = B[j][i];
```

```
for (int i=0; i<N-N%m; i+=m){
    for (int j=0; j<N; ++j)
        A[i][j] = B[j][i];
    for (int j=0; j<N; ++j)
        A[i+1][j] = B[j][i+1];
    // ...
    for (int j=0; j<N; ++j)
        A[i+m-1][j]=B[j][i+m-1];
}
// resíduo do laço (loop remainder)
for (int i=N-N%m; i<N; ++i)
    for (int j=0; j<N; ++j)
        A[i][j] = B[j][i];
```

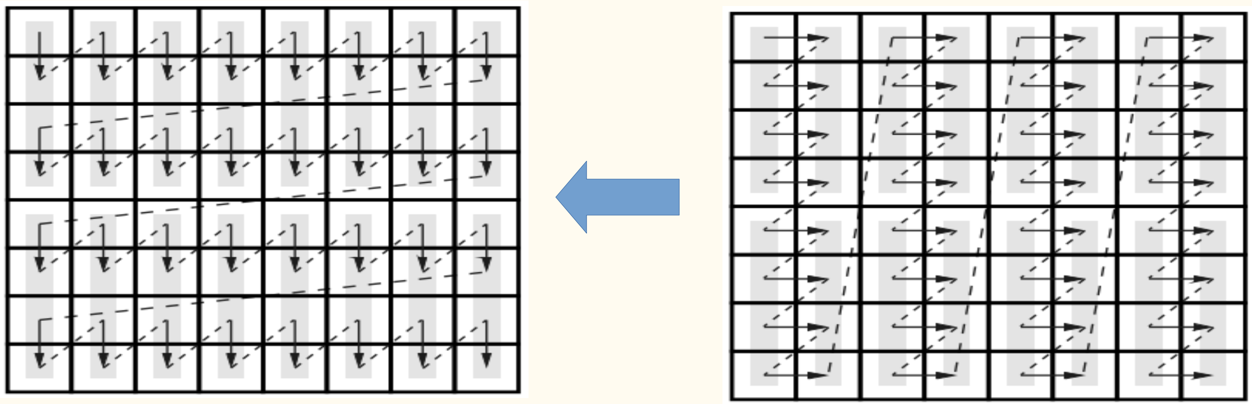
- Fator de *unroll*: **m**
- Nenhum ganho de performance apenas com *unroll*

Transposta → Unroll & Jam

```
for (int i=0; i<N-N%m; i+=m)
    for (int j=0; j<N; ++j){
        A[i][j] = B[j][i];
        A[i+1][j] = B[j][i+1];
      // ...
        A[i+m-1][j] = B[j][i+m-1];
   }
// resíduo do laço
for (int i=N-N%m; i<N; ++i)
    for (int j=0; j<N; ++j)
        A[i][j] = B[j][i];
```

Transposta → Loop Unroll & Jam (m=2, N=8)

- Transposta de matriz com *unroll* de fator 2



A[N][N]   B[N][N]

Transposta → Loop Unroll & Jam

- Ganhos de performance:
  - ➔ Linha de cache de **B** é carregada de qualquer forma, então pode ser reusada
- Ciladas de performance:
  - ➔ São necessários mais registradores (um para cada linha de **A**)
  - ➔ Mais linhas da matriz **A** em cache
  - ➔ Mais endereços no mapeamento de memória (*TLB misses*)
  - ➔ Risco de *cache thrashing*
  - ➔ *Reuso em matriz **B** aumenta problemas em **A***

Transposta → Loop Blocking

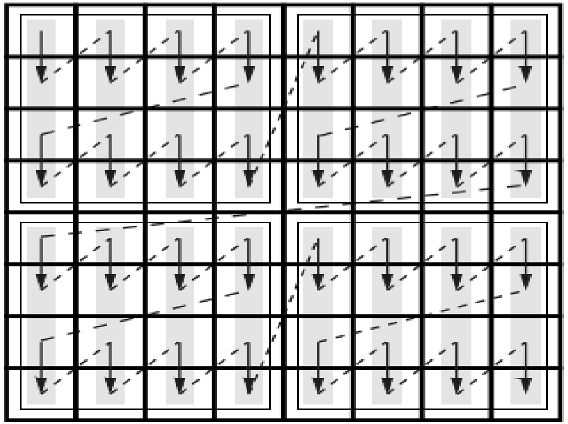
- *Unroll* em laços grandes aumenta pressão em registradores e esgota capacidade da *TLB*

- Blocagem de laço (*Loop Blocking*) aumenta reutilização da cache

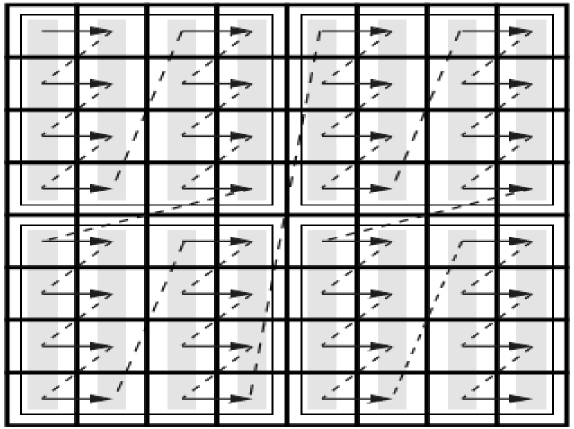
  ➔ A ideia é reduzir o problema para tamanhos que caibam na cache

  ➔ Permite o uso completo de linhas de cache já carregadas sem pressão adicional em registradores

Transposta → *U&J* + *Blocking* (m=2, b=4, N=8)



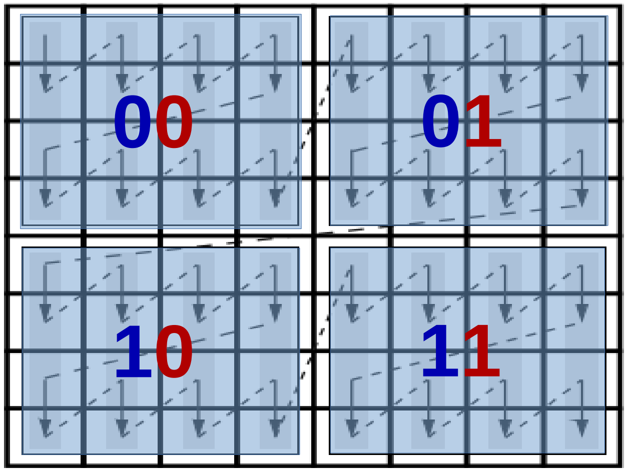
A[N][N]



B[N][N]

Transposta → U&J + Blocking (m=2, b=4, N=8)

```
for (int ii=0; ii<N/b; ++ii){
  istart=ii*b; iend=istart+b;
  for (int jj=0; jj<N/b; ++jj) {
    jstart=jj*b; jend=jstart+b;
    for (int i=istart; i<iend; i+=m)
      for (int j=jstart; j<jend; ++j){
           A[i][j] = B[j][i];
           A[i+1][j] = B[j][i+1];
           // ...
           A[i+m-1][j] = B[j][i+m-1];
      }
  }
}
```



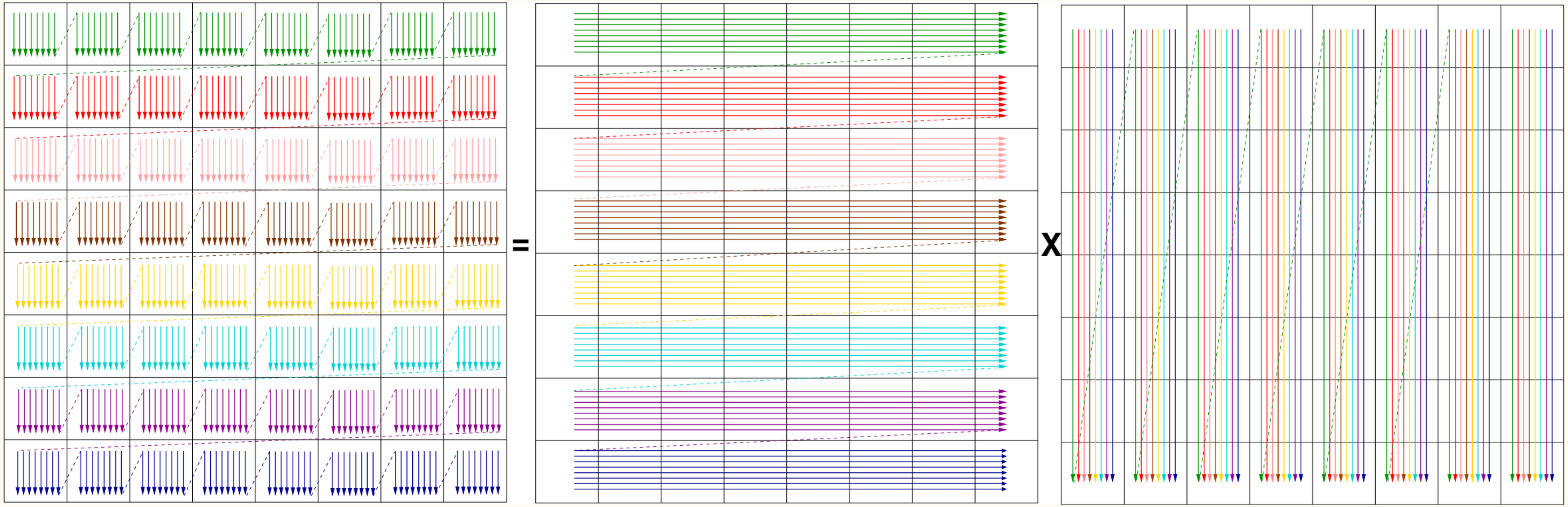
Multiplicação Matriz-matriz (MMM)

- MMM → **C = A * B**

```
for (int i=0; i<N; ++i){
   for (int j=0; j<N; ++j)
     for (int k=0; k<N; ++k)
         C[i][j] += A[i][k] * B[k][j];
```

- Cada linha de **A** carregada **N** vezes
- Cada coluna de **B** carregada **N** vezes (acesso *strided*)
- Elementos de **C** carregados uma vez

Multiplicação Matriz-matriz (MMM)

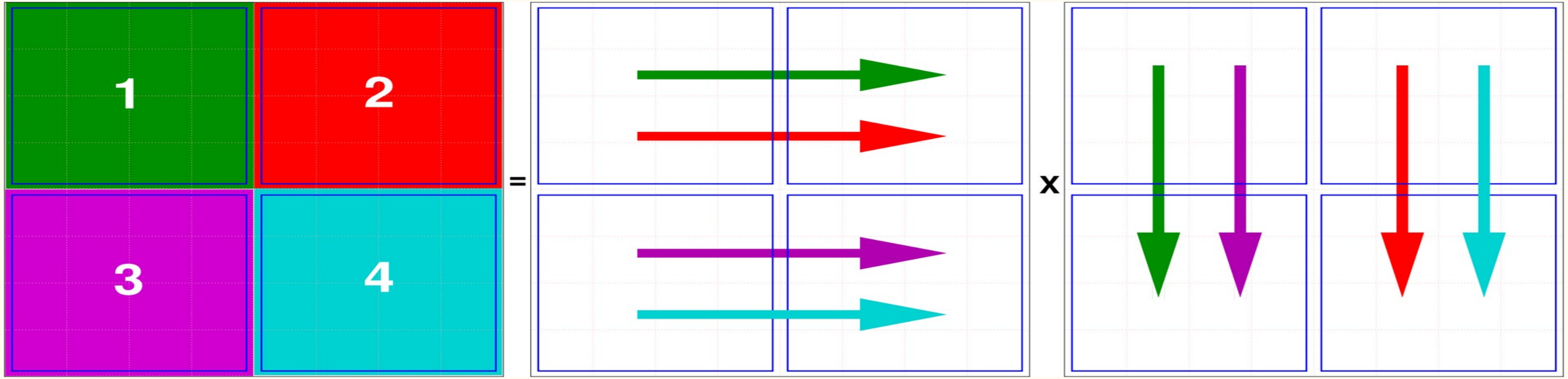


C = A x B

MMM → Unroll & jam

```
for (int i=0; i<N; ++i){
  for (int j=0; j<N-N%m; j+=m) {
    C[i][j] = C[i][j+1] =...= C[i][j+m-1]=0.0;
    for (int k=0; k<N; ++k) {
      C[i][j] += A[i][k]*B[k][j];
      C[i][j+1] += A[i][k]*B[k][j+1];
      ...
      C[i][j+m-1] += A[i][k]*B[k][j+m-1];
    }
  }
  // resíduo do laço j
  for (int j=N-N%m; j<N; ++j) {
    C[i][j] = 0.0;
    for (int k=0; k<N; ++k)
      C[i][j] += A[i][k]*B[k][j];
  }
}
```

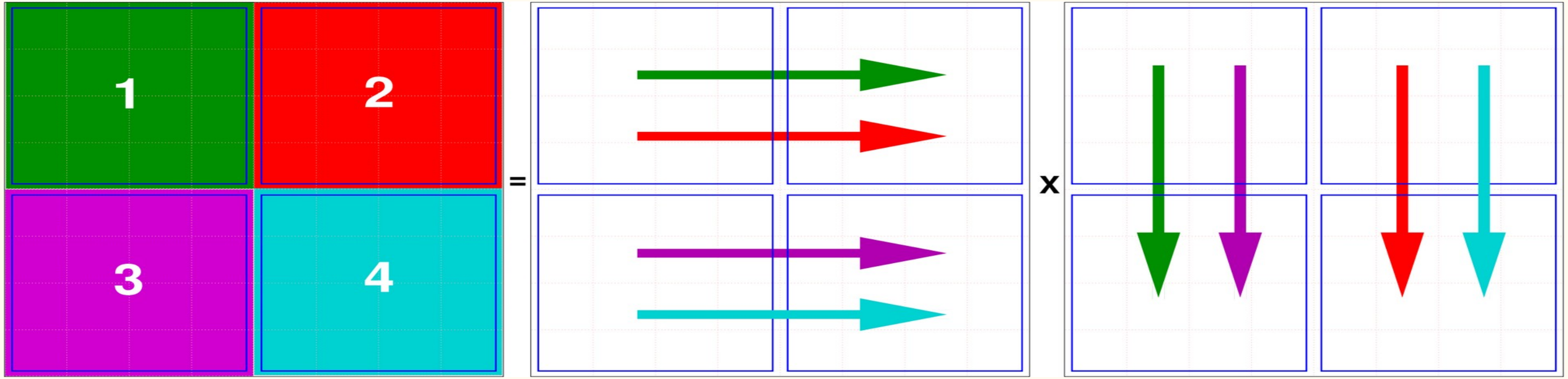
MMM → *Loop blocking*



A = B x C

- Dividir matriz em blocos que caibam na Cache
- Cada bloco é computado com ***Unroll & Jam***
  - ➔ Considere tamanho de bloco (**b**) múltiplo de tamanho de *unroll* (**m**)
  - ➔ Considere tamanho de matriz (**N**) múltiplo de tamanho de bloco (**b**)

MMM → *Loop blocking*



A = B x C

- Linhas em **A** ainda estão em cache quando reusadas
- Bloco em **B** é carregado no primeiro acesso e permanece em cache
  - ➔ Cada valor em **B** é carregado **N/b** vezes em cache, ao invés de **N**

MMM com *loop blocking*

```
for (int ii=0; ii<N/b; ++ii) {
  istart=ii*b; iend=istart+b;
  for (int jj=0; jj<N/b; ++jj) {
    jstart=jj*b; jend=jstart+b;
    for (int kk=0; kk<N/b; ++kk) {
      kstart=kk*b; kend=kstart+b;
      for (int i=istart; i<iend; ++i)
        for (int j=jstart; j<jend; j+=m) {
          C[i][j] = C[i][j+1] =...= C[i][j+m-1]=0.0;
          for (int k=kstart; k<kend; ++k) {
            C[i][j] += A[i][k]*B[k][j];
            C[i][j+1] += A[i][k]*B[k][j+1];
            ...
            C[i][j+m-1] += A[i][k]*B[k][j+m-1];
          }
        }
    }
  }
}
```

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).