OTIMIZAÇÃO DE CÓDIGO POR COMPILADORES

- livro HIGH PERFORMANCE COMPUTING (Kevin Dowd, capítulo IV);
- Objetivo: fornecer respostas corretas e executar rapidamente;
- Código gerado pode ser maior, mas com menos instruções executáveis ou com mais instruções porém mais eficiente;
- Tipos de otimização:
  clássicas e dependentes do h/w;
- Compilação em Ling. Intermediária (LI);

```
A = -B + C * D / E


T1 := D / E
T2 := C * T1
T3 := -B
A := T3 + T2
```

Linguagem Intermediária

- instruções simples, próximas da **linguagem de máquina**;
- operador, operandos e resultado: **X := Y op Z**
- referência a variáveis temporárias (uso de registradores);
- tratamento de desvios: separação do cálculo do contador e goto's para endereços absolutos;
- idéia básica: formar BLOCOS BÁSICOS (cada um com uma única entrada e uma única saída) tão independentes quanto possível entre si e substituir os desvios por um GRAFO DE FLUXO;
- identificar, dentro de cada bloco, as variáveis definidas e as utilizadas;

Exemplo com um trecho de código (*loop*) escrito em C:

```
while (j < n) {
k = k + j*2;
m = j*2;
j++;
}
```

```
A::
t1 := j
t2 := n
t3 := t1 < t2
jmp (B) t3
jmp (C) TRUE
```

```
B::
t4 := k
t5 := j
t6 := t5 * 2
t7 := t4 + t6
k := t7
```

```
t8 := j
t9 := t8 * 2
m := t9
t10 := j
t11 := t10 + 1
j := t11
jmp (A) TRUE
```

```
C::
```

DAG - Directed Acyclic Graph

Possibilitam a análise das variáveis definidas e utilizadas.

- internos aos blocos;
- identificam as dependências entre variáveis;
- possibilita otimizar blocos (menor tempo de execução e menor uso de registradores);
- "top to bottom";

Exemplo do "loop" (bloco B revisto):

```
B::
t4 := k
t5 := j
t6 := t5 * 2
m := t6
t7 := t4 + t6
k := t7
t11 := t5 + 1
j := t11
jmp (A) TRUE
```

Análise do fluxo de dados entre blocos

- identificar as variáveis definidas e utilizadas em cada bloco;
- identificar variáveis independentes;
- eliminar expressões redundantes;
- mover atribuições de um bloco para outro;
- possibilita liberar recursos (menor uso de registradores e menor número de intruções);

Otimização de LOOPS

- importância de otimizar;
- identificar o loop (nó dominante e o caminho de retorno);
- remover instruções "invariantes";
- identificar trechos que utilizam o contador;
- exemplo;

Código referente à figura (grafo de fluxo).

```
      .....
      NNODES = 0                    A
      DO 10 I=1,N                   B
         J = LIST(I)                B
         IF (J.EQ.0) GOTO 30        B
20       J = NEXT(J)                C
         NNODES = NNODES + 1        C
30       IF (J.NE.0) GOTO 20        D
10    CONTINUE                      E
```

Geração do Código Objeto

- específico para o h/w;
- otimizações incluem a distribuição de recursos e escalonamento de intruções;
- otimizações por s/w (compilador) *versus* otimizações por h/w (processador);

- compilação para lingugem intermediária de nível 2;
- tentar prover maior paralelismo para arquiteturas multiprocessadas ou superescalares ou VLIW;
- remoção de “dead code”;
- exemplo: RISC-6000 e processadores atuais;
  - arquitetura Harvard;
  - auto-incremento de registradores;
  - h/w pre-fetching;
  - renomeamento de registradores;
  - execução “especulativa” (branch delay slot);

TÉCNICAS CLÁSSICAS DE OTIMIZAÇÃO

Entender o que o compilador pode e o que não pode fazer ajuda a programar melhor.

- Propagação de cópias (**copy propagation**, intra e inter-blocos).

```
X = Y
Z = 1. + X
```

```
X = Y
Z = 1. + Y
```

- Desdobramento de constantes (**constant folding** (calcula constantes em tempo de compilação).

```
INTEGER I,K
PARAMETER (I = 100)
k = 200
J = I + K
```

- Remoção de código inoperante (**dead code removal**).
  - Código inatingível ou que gera resultados que não serão empregados, gerado pelo autor do s/w ou por LI's.
  - Não há warning na otimização (somente na compilação normal);
- Redução do grau da operação (**strength reduction**).

```
Y = X**2
J = K*2
```

```
Y = X*X
J = K + K
```

- Renomeamento de variáveis (**variable renaming**), análoga ao renomeamento de registradores (register renaming), permite maior grau de paralelismo e clareza.

```
X = Y*Z;
Q = R + X + X;
X = A + B;
```

```
X0 = Y*Z;
Q = R + X0 + X0;
X = A + B;
```

- Eliminação de subexpressões comuns (**common subexpression elimination**) (dentro de uma rotina).

```
D = C * (A + B)
E = (A + B)/2
```

```
temp = A + B
D = C * temp
E = temp/2
```

Também aplicável ao cálculo de endereços:

```
address(A) + (I-1)*sizeof_datatype(A) +
+ (J-1)*sizeof_datatype(A) * column_dimension(A)
```

- Reposicionamento de código invariante para fora do loop (**loop invariant code motion**).

```
DO 10 I = 1,N
   A(I) = B(I) + C*D
   E = G(K)
10 CONTINUE

temp = C*D
DO I = 1,N
   A(I) = B(I) + temp
ENDDO
E = G(K)
```

- Simplificação no cálculo de variáveis induzidas (**induction variable simplification**).
  - variáveis induzidas são aquelas calculadas linearmente a partir do contador;
  - especialmente útil para processadores que provém incremento automático de registradores;

```
DO 10 I=1,N
   K = I*4 + M
10 CONTINUE

K = M
DO I=1,N
   K = K + 4
ENDDO
```

Outro exemplo:

```
DO I = 1,N
address = base_address(A)
          + (I-1) * sizeof_datatype(A)
ENDDO


address = base_address(A) - sizeof_datatype(A)
DO I = 1,N
address = address + sizeof_datatype(A)
ENDDO
```

- Detecção de variáveis residentes em registradores (**register variable detection**).
  - análise de fluxo de dados para determinar variáveis que podem ser residentes em registradores;
  - importante na arquitetura RISC (evitar load/store);

Considerações finais.

- benchmarking implica no maior grau de otimização possível;
- pre-compilador pode substituir chamadas a subrotinas por inserção do código correspondente para evitar mudanças de contexto (**inlining**);
- otimizações clássicas + dependentes de h/w;
- **linker** procura agrupar conjuntamente o programa e as rotinas no módulo de carga para minimizar o cache miss de instruções;

- otimizador post-linkage para realocar o uso de registradores e reordenar os módulos do programa para melhorar a utilização do cache;
- decisões **run-time** podem se sobrepor em relação às feitas em **compile-time**;

Exemplo de incremento de paralelismo de instrução *versus* redução de cache misses

```
for (i = 0; i < 512; i = i+1)
    for (j = 1; j < 512; j = j+1)
        x[i][j] = 2 * x[i][j-1];

for (i = 0; i < 512; i = i+1)
    for (j = 1; j < 512; j = j+4) {
        x[i][j]   = 2 * x[i][j-1];
        x[i][j+1] = 2 * x[i][j];
        x[i][j+2] = 2 * x[i][j+1];
        x[i][j+3] = 2 * x[i][j+2];
        };

for (j = 1; j < 512; j = j+1)
    for (i = 0; i < 512; i = i+4) {
        x[i][j]   = 2 * x[i][j-1];
        x[i+1][j] = 2 * x[i+1][j-1];
        x[i+2][j] = 2 * x[i+2][j-1];
        x[i+3][j] = 2 * x[i+3][j-1];
        };
```

CLAREZA DE CÓDIGO

- livro HIGH PERFORMANCE COMPUTING (Kevin Dowd, capítulo V);
- Preceitos de estilo;
- Comentários e separações;
- Escolha adeaquada de nomes de variáveis;
- Variáveis em FORTRAN;
- Uso de constantes (PARAMETER, # define):

```
C**********************************
INTEGER NX
REAL EPSILON
PARAMETER (EPSILON = 1.0E-38, NX = 100);
C**********************************
```

- Compatibilidade nas atribuições de valores a variáveis:

```
C*************************************
      double precision A
      A = 1/3
      A = 1./3.
      A = 1.0d0/3.0d0
C*************************************
```

- Uso de COMMON (FORTRAN):

```
C*************************************
      IMPLICIT NONE
      INTEGER MAXPTS
      PARAMETER (MAXPTS = 1000)
      REAL X(MAXPTS), Y(MAXPTS)
      COMMON /POINTS/ X, Y
C*************************************
```

- Uso de INCLUDE (C & FORTRAN);

- Forma de armazenamento dos dados:

```
C**********************************
PROGRAM MAIN
INTEGER M,N
PARAMETER (N=3, M=4)
DOUBLE PRECISION C(N,M)
CALL FAZ (C, N*M)
END
C**********************************
SUBROUTINE FAZ(C,K)
DOUBLE PRECISION C(K)
DO I = 1,K
   C(K) = 0.0D0
ENDDO
C**********************************
```

CONSIDERAÇÕES SOBRE PORTABILIDADE DE CÓDIGO

- livro HIGH PERFORMANCE COMPUTING (Kevin Dowd, capítulo VI);
- "Funcionava e deixou de funcionar..." (comum quando código fora dos padrões da linguagem é portado para arquiteturas RISC);
- Aliasing (ANSI FORTRAN):

```
C************************************
REAL A, B, C
INTEGER I
EQUIVALENCE (A,I)
B = A
CALL FOO (A,A)
CALL BAR (A,B)
CALL BAZ (A,I)
C************************************
PROGRAM MAIN
INTEGER IX = 1
CALL ADD1 (IX,IX)
WRITE (*,*) IX
END
```

```
SUBROUTINE ADD1 (IA,IB)
INTEGER IA, IB
...     ...
IF (IA.EQ.1) IB = IA +1
IF (IB.EQ.2) IA = 1
END
C************************************
```

- Incompatibilidade de tipos de variáveis (exemplo IEEE-754);
- Problemas de armazenamento devido a tipos diferentes de variáveis:
  - uso do equivalence;
  - uso do union:

```
...
 union {
        int i[2];
        double x;
 } foo;
...
```

- uso de ferramentas de alocação automática de memória;

- Restrições de alinhamento de referências à memória:

```
---------------------------------------
TIPO DE DADO              ALINHAMENTO
                    (endereço múltiplo de)
---------------------------------------
double precision           8 byte (8)
integer, real, logical     4 byte (4)
integer*2                  2 byte (2)
byte, character            1 byte (-)
---------------------------------------
```

```
C************************************
...
REAL*4 A, B
REAL*8 R
COMMON /BAR/ A, R, B
...
C************************************
```

```
PROGRAM BAR
INTEGER IARRAY (100000)
COMMON /STUFF/ IARRAY
CALL FOO (IARRAY(2),49999)
END

SUBROUTINE FOO (DPVAL,N)
DOUBLE PRECISION DPVAL(50000)
DO 10 I = 1, N
   DPVAL(I) = 1.0D0
10 CONTINUE
END
C**************************************
----------------------------------------
main ()
{
  int *p;
  double *q;
  int i;

  p = (int *) malloc (sizeof(int) * 3);
  q = (double *) (p + 1);

  for (i=0; i<100000; i++)
      *q = (double) 1.0;
}
----------------------------------------
```

(mas tipos diferentes podem conviver numa struct)

TEMPORIZAÇÃO & PROFILING

- Definição e necessidade de *timing/profiling*.
- Erros devido à frequência de amostragem.
- Temporização e profiling devem ser feitos para todas as instâncias de dados de entrada.
- Otimização: *timing/profiling* + *code tuning*;
- Técnicas e ferramentas:
  1. Comando **time** (UNIX/LINUX)
  2. Temporização de partes do código através de rotinas do sistema operacional.
  3. *Profiling*.
  4. Uso de contadores de *H/W* da CPU (clock, L1/L2 misses, flops, TLB misses, etc.).

COMANDO TIME E CONCEITOS DE CPU, USER E KERNEL TIME

- Comando **time** (UNIX/LINUX): CPU time & elapsed time.
- **CPU time = user time + kernel time**
- **kernel/system time** deve-se a chamadas do processo do usuário a rotinas do sistema operacional (inclui page faults, referências desalinhadas à memória, tratamento de exceções, etc.).
- quando **elapsed time** >>> **cpu time**, tem-se alto grau de multiprogramação (por ex. muitos processos de usuários), muito **input/output**, relativo a tempo de acesso ao disco rígido e outros periféricos (terminais, dispositivos de fitas, etc.).

PROFILING

- Perfis de execução e a Lei de Amdahl.
- Ferramentas mais comuns (UNIX/LINUX): **prof** e **gprof**
- Profilers para blocos básicos:
  - **tcov** (SunOS & Solaris);
  - **lprof** (RISC 6000 e outros UNIX's);
  - **pixie**;
- Uso do **prof**:

```
% cc loops.c -p -o loops
% ./loops
% prof loops
```

```
#include <math.h>
main () {
    int i;
    for (i=0;i<100;i++) {
        if (i == 2*(i/2)) loop1 ();
        loop2 ();
        loop3 ();
        loop4 ();       }
}
loop1 () {
      int i;
      double x[10000];
      for (i=0;i<10000;i++)
          x[i] = x[i]*x[i];
}
loop2 () {
      int i;
      double x[50000];
      for (i=0;i<50000;i++)
          x[i] = x[i]*x[i];
}
loop3 () {
      int i;
      double x[100000];
      for (i=0;i<100000;i++)
          x[i] = x[i]*x[i];

}
```

```
loop4 () {
      int i;
      double x[100000];
      for (i=0;i<100000;i++)
          x[i] = pow(x[i],3.5);
}
```

```
*************************************************
Name       %Time  Seconds  Cumsecs #Calls   msec/call
.pow        47.6    30.53   30.53  10000000    0.0031
._mcount    21.4    13.71   44.24
.loop4      12.0     7.69   51.93       100   76.90
.loop3       9.4     6.06   57.99       100   60.60
.loop2       4.7     3.02   61.01       100   30.20
.loop1       0.4     0.28   63.95        50    5.6
            ...
.main        0.0     0.00   64.18         1    0.
*************************************************
```

- Uso do **gprof**:

```
% cc loops.c -pg -o loops
% ./loops
% gprof loops
```

```
                                called/total       parents
index  %time    self descendents  called+self    name           index
                                called/total       children

                0.00       44.45       1/1           .__start [2]
[1]     53.0    0.00       44.45       1          .main [1]
                6.03       37.65     100/100         .loop3 [3]
                0.50        0.00     100/100         .loop2 [10]
                0.27        0.00      50/50          .loop1 [17]

-----------------------------------------------

                6.03       37.65     100/100         .main [1]
[3]     52.1    6.03       37.65     100          .loop3 [3]
                7.13       30.52     100/100         .loop4 [4]

-----------------------------------------------

                7.13       30.52     100/100         .loop3 [3]
[4]     44.9    7.13       30.52     100          .loop4 [4]
               30.52        0.00 10000000/10000000     .pow [6]

-----------------------------------------------

               30.52        0.00 10000000/10000000     .loop4 [4]
[6]     36.4   30.52        0.00 10000000          .pow [6]
                0.00        0.00     100/100         .loginner2 [20]
                0.00        0.00     100/100         .expinner2 [19]

-----------------------------------------------

                0.50        0.00     100/100         .main [1]
[10]     0.6    0.50        0.00     100          .loop2 [10]

-----------------------------------------------

                0.27        0.00      50/50          .main [1]
[17]     0.3    0.27        0.00      50          .loop1 [17]

-----------------------------------------------
```

CONCEITOS DE PARALELISMO PARA MÁQUINAS MONOPROCESSADAS

- Otimizações possíveis: aumentar o grau de paralelismo, melhorar acesso à memória, utilizar algoritmos mais eficientes.
- Grau de paralelismo: para máquinas monoprocessadas, relaciona-se com o número de instruções que podem ser executadas concorrentemente (num pipeline) ou simultaneamente (em unidades funcionais diferentes) pela CPU. Fala-se em paralelismo em nível de instrução, ou em granulação fina.
  - Aumentar o paralelismo significa eliminar dependências de dados e de instruções.
  - "Ajudar" o compilador a reconhecer código paralelizável, deixando mais claras as não-dependências.
  - Não há receitas prontas.

  - Técnicas diferentes para processadores escalares e vetoriais;
  - Arquitetura RISC: paralelismo graças a pipelines.
  - Aritmética vetorial: paralelismo graças a operações com vetores/matrizes.

```
DO I = 1, N
   A(I) = B(I) * C
ENDDO
```

$\mathbf{A} \leftarrow \mathbf{A} + \mathbf{B} * \mathrm{C}$

- Supercomputação: processadores vetoriais *versus* máquinas paralelas.
- Programar evitando dependências de dados e de controle, inclusive para poder portar o código para máquina paralela.

Dependência de dados

```
   DO 10 I = 1,N
      A(I) = A(I) + B(I)
10 CONTINUE
      A(I)   = A(I)   + B(I)
      A(I+1) = A(I+1) + B(I+1)
      A(I+2) = A(I+2) + B(I+2)
```

- dependência de fluxo (ou *backwards*):

```
      DO 10 I = 2,N
         A(I) = A(I-1) + B(I)
   10 CONTINUE


         A(I)   = A(I-1) + B(I)
         A(I+1) = A(I)   + B(I+1)
         A(I+2) = A(I+1) + B(I+2)


      DO 10 I = 2,N,2
         A(I)   = A(I-1) + B(I)
         A(I+1) = A(I-1) + B(I) + B(I+1)
   10 CONTINUE
```

- anti-dependências:

```
      X = A/B
      Y = X + 2.0
      X = D - E

      temp = A/B
      Y    = temp + 2.0
      X    = D - E
---------------------------------------
   DO 10 I = 1,N
      A(I) = B(I)*E
      B(I) = A(I+2)*C
10 CONTINUE

Vector processor:

      A(I)   = B(I)*E
      A(I+1) = B(I+1)*E
      A(I+2) = B(I+2)*E
          ...
      B(I)   = A(I+2)*C  (erro !!!)
      B(I+1) = A(I+3)*C
      B(I+2) = A(I+4)*C

(seria necessário utilizar variáveis temporárias
 para vetorizar)
```

```
RISC processor:

      A(I)   = B(I)*E
      B(I)   = A(I+2)*C
      A(I+1) = B(I+1)*E
      B(I+1) = A(I+3)*C
      A(I+2) = B(I+2)*E
      B(I+2) = A(I+4)*C
```

- dependências de definição
  (*output dependencies*)

```
   DO 10 I = 1,N
      A(I)   = C(I)*2.
      A(I+2) = D(I) + E
10 CONTINUE


      A(I)   = C(I)*2.
      A(I+1) = C(I+1)*2.
      A(I+2) = C(I+2)*2.
      A(I+2) = D(I) + E     (erro !!!)
      A(I+3) = D(I+1) + E
      A(I+4) = D(I+2) + E
```

Distância de Dependência:

```
< 0  dependência de fluxo
= 0  não-dependência
> 0  anti-dependência
```

```
   DO 10 I = 1,N
      A(I) = B(I)*E
      B(I) = A(I+2)*C
10 CONTINUE
```

"Referências Ambíguas"

```
   DO 10 I = 1,N
      A(I) = B(I)*E
      B(I) = A(I+K)*C   (K desconhecido)
10 CONTINUE
```

Dependências de Controle

- Reposicionamento de instruções "custosas" (exemplo).
- Branch delay slot.

OTIMIZAÇÃO DE LOOPS

Loop "unrolling":

```
DO I=1,N
   A(I) = A(I) + B(I) * C
ENDDO
```

(incluir loop pré-condicionador)

```
II = IMOD(N,4)
DO I = 1, II
   A(I) = A(I) + B(I) * C
ENDDO

DO I = 1+II,N,4
   A(I)   = A(I)   + B(I)   * C
   A(I+1) = A(I+1) + B(I+1) * C
   A(I+2) = A(I+2) + B(I+2) * C
   A(I+3) = A(I+3) + B(I+3) * C
ENDDO
```

O "unrolling" pode não ser conveniente:

- poucas iterações (contador baixo);
- loops com muitas intruções (*fat loops*);
- loops com chamadas de subrotinas;

```
DO I = 1, N
   CALL SHORT (A(I), B(I), C(I))
ENDDO
   ...
SUBROUTINE SHORT (A,B,C)
A = A + B * C
RETURN
END
```

```
DO I = 1+II, N, 4
   A(I)   = A(I) + B(I) * C
   A(I+1) = A(I+1) + B(I+1) * C
   A(I+2) = A(I+2) + B(I+2) * C
   A(I+3) = A(I+3) + B(I+3) * C
ENDDO
```

"Unrolling" depende de se fazer **inlining** (rotinas curtas). Podem ocorrer funções ou

macros que não geram chamadas a subrotinas, sendo substituídas por linhas do código correspondente.

- **loops com desvios** - unrolling pode ser eficiente quando a CPU tiver processador de desvio. Exemplo:

```
DO I = 1, N
   DO J = 1, N
      IF (B(J,I).GT.1.0)
         A(J,I) = A(J,I) + B(J,I)*C
   ENDDO
ENDDO

DO I = 1, N
   DO J = II+1, N, 4
      IF (B(J,I).GT.1.0)
         A(J,I)   = A(J,I) + B(J,I)*C
      IF (B(J+1,I).GT.1.0)
         A(J+1,I) = A(J+1,I) + B(J+1,I)*C
      IF (B(J+2,I).GT.1.0)
         A(J+2,I) = A(J+2,I) + B(J+2,I)*C
      IF (B(J+3,I).GT.1.0)
         A(J+3,I) = A(J+3,I) + B(J+3,I)*C
   ENDDO
ENDDO
```

- **loops recursivos** - Há dependência de fluxo e pode-se fazer unrolling às custas de mais operações.

```
DO I = 2, N
   A(I) = A(I) + A(I-1)*B
ENDDO

A(I)   = A(I) + A(I-1)*B
A(I+1) = A(I+1) + A(I)*B
A(I+2) = A(I+2) + A(I+1)*B
A(I+3) = A(I+3) + A(I+2)*B

DO I = 2, N, 2
A(I+1) = A(I+1) + (A(I) + A(I-1)*B)*B
  A(I) = A(I)   + A(I-1)*B
ENDDO
```

- Problemas em potencial:
  - escolha do fator de "unrolling";
  - *register thrashing*;
  - *instruction cache miss*;
  - limitações de *memory bandwidth* em máquinas multiprocessadas;

Loop "unrolling" de loops externos:

```
/* loop triplo original */

for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            a[i][j][k] = a[i][j][k] + b[i][j][k]


/* unrolling do loop intermediário */

for (i=0; i<n; i++)
    for (j=0; j<n; j+=2)
        for (k=0; k<n; k++)
            a[i][j][k] = a[i][j][k]     + b[i][j][k]
            a[i][j+1][k] = a[i][j+1][k] + b[i][j+1][k]

/* unrolling dos loops intermediário e interno */

for (i=0; i<n; i++)
    for (j=0; j<n; j+=2)
        for (k=0; k<n; k+=2)
a[i][j][k]     = a[i][j][k]       + b[i][j][k]
a[i][j+1][k]   = a[i][j+1][k]     + b[i][j+1][k]
a[i][j][k+1]   = a[i][j][k+1]     + b[i][j][k+1]
a[i][j+1][k+1] = a[i][j+1][k+1]   + b[i][j+1][k+1]
```

Exemplo - unrolling do loop externo devido a este ter maior número de iterações
(no caso, $N >>> M$):

```
DO I = 1, N
   DO J = 1, M
      A(J,I) = B(J,I) + C(J,I)*D
   ENDDO
ENDDO

II = IMOD(N,4)
DO I = 1, II
   DO J = 1, M
      A(J,I) = B(J,I) + C(J,I)*D
   ENDDO
ENDDO

DO 10 I = II+1, N, 4
   DO 20 J = 1, M
      A(J,I) = B(J,I) + C(J,I)*D
      A(J,I+1) = B(J,I+1) + C(J,I+1)*D
      A(J,I+2) = B(J,I+2) + C(J,I+2)*D
      A(J,I+3) = B(J,I+3) + C(J,I+3)*D
   ENDDO
ENDDO
```

Exemplo - unrolling do loop externo devido à dependência de dados entre iterações do loop interno:

```
DO J = 1, M
   DO I = 2, N
      A(I,J) = A(I,J) + A(I-1,J)*D
   ENDDO
ENDDO

C Loop condicionador

JJ = IMOD(M,4)
DO J = 1, JJ
   DO I = 2, N
      A(I,J) = A(I,J) + A(I-1,J)*D
   ENDDO

DO J = 1+JJ, M, 4
   DO I = 2, N
      A(I,J)   = A(I,J)   + A(I-1,J)*D
      A(I,J+1) = A(I,J+1) + A(I-1,J+1)*D
      A(I,J+2) = A(I,J+2) + A(I-1,J+2)*D
      A(I,J+3) = A(I,J+3) + A(I-1,J+3)*D
   ENDDO
```

TRANSFORMAÇÕES ASSOCIATIVAS:

(X + Y) + Z equivale a (Y + X) + Z OK

Mas, (Y + Z) + X ???

Considere: X = Y = .00005 e Z = 1.0000

- **Reduções (por ex. produto escalar)**:

```
SUM = 0.0
DO I = 1, N
   SUM = SUM + A(I) * B(I)
ENDDO

C Loop unrolling para RISC
C RISC's suportam  r1*r2 + r3 -> r4

SUM0 = 0.0
SUM1 = 0.0
SUM2 = 0.0
SUM3 = 0.0
DO I = 1, N, 4
   SUM0 = SUM0 + A(I) * B(I)
   SUM1 = SUM1 + A(I+1) * B(I+1)
   SUM2 = SUM2 + A(I+2) * B(I+2)
   SUM3 = SUM3 + A(I+3) * B(I+3)
ENDDO
SUM = SUM0 + SUM1 + SUM2 + SUM3
```

```
C Loop unfolding para proc. vetorial

DO I = 1, N
   TEMP(I) = A(I) * B(I)
ENDDO

DO I = 1, N/2
   TEMP(I) = TEMP(I) + TEMP(I+N/2)
ENDDO
DO I = 1, N/4
   TEMP(I) = TEMP(I) + TEMP(I+N/4)
ENDDO
DO I = 1, N/8
   TEMP(I) = TEMP(I) + TEMP(I+N/8)
ENDDO

X = 0
DO I = 1, N/8
   X = X + TEMP(I)
ENDDO
```

- **Daxpy's**: (2 flops & 3 load/stores, não conveniente para RISC's)

```
    DO 10 I = 1, N
10     DY(I) = DY(I) + DA * DX(I)
```

- **Multiplicação de matrizes:**

```
C Inicializar com zero os elementos C(J,I)

DO K = 1, N
   DO J = 1, N
      DO I = 1, N
         C(I,J) = C(I,J) + A(I,K)*B(K,J)
      ENDDO
   ENDDO
ENDDO



C PASSO I: Permutar os loops mais externos
C Inicializar com zero os elementos C(J,I)

DO J = 1, N
   DO I = 1, N
      DO K = 1, N
         C(I,J) = C(I,J) + A(I,K)*B(K,J)
      ENDDO
   ENDDO
ENDDO

C PASSO II: Loop unrolling
C Inicializar com zero os elementos C(J,I)
```

```
DO J = 1, N
   DO I = 1, N

      KK = IMOD (N,4)
      DO K = 1, KK
         C(I,J) = C(I,J) + A(I,K)*B(K,J)
      ENDDO

      TEMP0 = 0.0
      TEMP1 = 0.0
      TEMP2 = 0.0
      TEMP3 = 0.0
      DO K = 1+KK, N, 4
         TEMP0 = TEMP0 + A(I,K)*B(K,J)
         TEMP1 = TEMP1 + A(I,K+1)*B(K+1,J)
         TEMP2 = TEMP2 + A(I,K+2)*B(K+2,J)
         TEMP3 = TEMP3 + A(I,K+3)*B(K+3,J)
      ENDDO
      C(I,J) = C(I,J)+TEMP0+TEMP1+TEMP2+TEMP3
   ENDDO
ENDDO
```

- **Permutação de loops e dependência de dados**:

```
C Seria conveniente o loop em K mais externo
C para permitir unrolling dos mais internos.
```

```
PARAMETER (IDIM=1000,JDIM=1000,KDIM=4)

DO I = 1, IDIM
   DO J = 1, JDIM
      DO K = 1, KDIM
         D(K,J,I) = D(K,J,I) + V(K,J,I)*DT
      ENDDO
   ENDDO
ENDDO

C Exemplo de possíveis dependências de dados:
DO I = 1, N-1
   DO J = 2, N
      A(I,J) = A(I+1,J-1) * B(I,J)
      C(I,J) = B(J,I)
   ENDDO
ENDDO
```

- **Balanço de load/stores e flops**:

```
C Exemplo de 3:1

DO I = 1, N
   A(I,J,K) = A(I,J,K) + B(I,J,K)
```

```
ENDDO

C Exemplo de 2:1

DO I = 1, N
   A(I) = A(I) + B(J)
ENDDO

C Exemplo de 6:6 (?)

for (i=0; i<n; i++) {
    xr[i] = xr[i]*yr[i] - xi[i]*yi[i]
    xi[i] = xr[i]*yi[i] + xi[i]*yr[i] }
```

- **Finalizando: otimização de loops**:
  - tornar loops vetorizáveis (eliminar desvios, inlining de rotinas curtas, permutar loops, etc.);
  - aumentar grau de paralelismo (loop unrolling, rearranjo de reduções, etc.);

ELIMINANDO A "BAGUNÇA":

- evitar chamadas a subrotinas/procedures (agravado pelo uso de variáveis comuns - COMMON, EXTERNAL, etc.);
- Uso de macros (pode requerer *pre-processor*:

```
#define average(x,y) ((x+y)/2)
```

- *inlining*:
  - lista na linha de comando do compilador
  - "diretivas "no programa fonte
  - automático
- evitar desvios/testes dentro de loops;
- evitar/otimizar testes longos:

```
if(((~i&j) && a<=0. && m==-1) || k==0)
```

```
if(k==0 || ((~i&j) && a<=0. && m==-1))
```

- evitar testes redundantes:

```
if (c=='A') A = TRUE;
if (c=='B') B = TRUE;
if (c=='C') C = TRUE;

switch (c) {
  case 'A': A = TRUE; break;
  case 'B': B = TRUE; break;
  case 'C': C = TRUE; break;
  }
```

- **Testes/desvios dentro de loops**:

```
C Não valeria a pena:
PARAMETER (SMALL = 1.E-20)

DO I = 1, N
   IF (ABS(A(I).GE.SMALL) THEN
      B(I) = B(I) + A(I)*C
   ENDIF
ENDDO

C Outro exemplo: código ineficiente
DO I = 1, N
   IF(A(I).LT.0.) A(I) = -A(I)
ENDDO
```

```
C O mesmo código otimizado:
#define ABS(Q) (Q < 0.? -Q : Q)
...
for (i=0; i<n; i++)
    a[i] = ABS(a[i]);
C Cuidado: C fabs() gera chamada...
```

- **Ainda testes/desvios dentro de loops**:
  - testes invariantes (vão para fora do loop):

```
DO I = 1, K
   IF (N.EQ.0) THEN
      A(I) = A(I) + B(I) *C
   ELSE
      A(I) = 0.
   ENDIF
ENDDO

IF (N.EQ.0) THEN
   DO I = 1, K
      A(I) = A(I) + B(I) *C
   ENDDO
ELSE
   DO I = 1, K
      A(I) = 0.
   ENDDO
ENDIF
```

– testes dependentes do contador podem desaparecer, substituídos por novos loops com contadores apropriados:

```
DO I = 1, N
   DO J = 1, N
      IF (J.LT.I) THEN
         A(J,I) = A(J,I)*C
      ELSE
         A(J,I) = 0.0
      ENDIF
   ENDDO
ENDDO

C Note que o loop em J com o IF-THEN-ELSE foi
C logicamente substituído por 2 novos loops

DO I = 1, N

   DO J = 1, I-1
      A(J,I) = A(J,I)*C
   ENDDO

   DO J = I,N
      A(J,I) = 0.0
   ENDDO

ENDDO
```

– testes sem dep. dados entre iterações:

```
DO I = 1, N
   DO J = 1, N
      IF (B(J).GT.1.0) THEN
         A(J,I) = A(J,I) + B(J,I)*C
      ENDIF
   ENDDO
ENDDO

C Pode-se fazer o loop unrolling:

DO I = 1, N
   DO J = 1, N, 2
      IF (B(J).GT.1.0) THEN
         A(J,I) = A(J,I) + B(J,I)*C
      ENDIF
      IF (B(J+1).GT.1.0) THEN
         A(J+1,I) = A(J+1,I) + B(J+1,I)*C
      ENDIF
   ENDDO
ENDDO
```

– testes com dep. dados entre iterações:

```
C DEVEM SER EVITADOS!
DO I = 1, N
   IF (X.LT.A(I)) X = X + B(I)*2.
ENDDO
```

- Exemplo com unrolling de uma redução:

```
for (i=0; i<n; i++)
    z = a[i] > z ? a[i] : z;

z0 = 0.;
z1 = 0.;

for (i=0; i<n-1; i+=2) {
    z0 = a[i] > z0 ? a[i]   : z0;
    z1 = a[i+1] > z1 ? a[i+1] : z1;
}
z = z0 < z1 ? z1 : z0;
```

- Conversões de tipos de dados:

```
C "Promoção" de B de REAL*4 para REAL*8
C  consome tempo

REAL*8 A(20)
REAL*4 B(20)
DO I = 1, 20
   A(I) = A(I) + B(I)
ENDDO
```

- Eliminação manual de expressões comuns:

```
/* nem tudo é óbvio para o compilador: */

d = a + b + c;
e = c + b + a;

temp = a + b + c;  ???

/* chamadas a idênticas a rotinas nunca são
   substituídas automaticamente: */

x = r*sin(a)*cos(b);
y = r*sin(a)*sin(b);
```

- "Aliviando" o loop ("loop invariant code motion" - exemplo para recuperar último caracter de um string):

```
while (*p != ' ')
      c = *p++;

while (*p++ != ' ');
      c = *(--p);
```

- Evitando load/store's inúteis, através do uso de variáveis temporárias:

```
/* bom para RISC, ruim para proc. vetorial */

for (i=0; i<n; i++) {
    xold[i] = x[i];
    x[i] = x[i] + xinc[i];
}

for (i=0; i<n; i++) {
    temp = x[i];
    xold[i] = temp;
    x[i] = temp + xinc[i];
}
```

- Otimização de acesso à memória: acesso por linhas (em C) ou por colunas (FORTRAN);
- Otimização de acesso à memória: um exemplo de "blocking":

```
DO I = 1, N
   DO J = 1, N
      A(J,I) = A(J,I) + B(I,J)
   ENDDO
ENDDO
```

```
C "blocking" - unrolling conjunto:
DO I = 1, N, 2
   DO J = 1, N, 2
      A(J,I)     = A(J,I)     + B(I,J)
      A(J+1,I)   = A(J+1,I)   + B(I,J+1)
      A(J,I+1)   = A(J,I+1)   + B(I+1,J)
      A(J+1,I+1) = A(J+1,I+1) + B(I+1,J+1)
   ENDDO
ENDDO

C otimizando o uso da TLB:
DO I = 1, N, 2
   DO J = 1, N/2, 2
      A(J,I)     = A(J,I)     + B(I,J)
      A(J+1,I)   = A(J+1,I)   + B(I,J+1)
      A(J,I+1)   = A(J,I+1)   + B(I+1,J)
      A(J+1,I+1) = A(J+1,I+1) + B(I+1,J+1)
   ENDDO
ENDDO
DO I = 1, N, 2
   DO J = N/2+1, N, 2
      A(J,I)     = A(J,I)     + B(I,J)
      A(J+1,I)   = A(J+1,I)   + B(I,J+1)
      A(J,I+1)   = A(J,I+1)   + B(I+1,J)
      A(J+1,I+1) = A(J+1,I+1) + B(I+1,J+1)
   ENDDO
ENDDO
```

```
C "blocking" para matrizes muito grandes:

II = MOD (N,16)
JJ = MOD (N,4)

DO I = 1, N
   DO J = 1, JJ
      A(J,I) = A(J,I) + B(I,J)
   ENDDO
ENDDO

DO I = 1, II
   DO J = JJ+1, N
      A(J,I) = A(J,I) + B(I,J)
   ENDDO
ENDDO

DO I = II+1, N, 16
   DO J = JJ+1, N, 4
      DO K = I, I+15
         A(J,K)   = A(J,K)   + B(K,J)
         A(J+1,K) = A(J+1,K) + B(K,J+1)
         A(J+2,K) = A(J+2,K) + B(K,J+2)
         A(J+3,K) = A(J+3,K) + B(K,J+3)
      ENDDO
   ENDDO
ENDDO
```

- Referências ambíguas/indiretas à memória:
  - Índices ambíguos de vetores ou matrizes dentro de loops: $A(I + K)$;
  - Referências indiretas: $A(K(I))$;
  - Indefinições na alocação de memória:

```
/* alocação bem definida: */
double a[N][N], c[N][N], d;
...
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        a[i][j] = a[i][j] + c[j][i]*d;

/* possibilidade de aliasing (a & c) */
double *a[n], *c[N], d;
...
for (i=0; i<N; i++) {
  a[i] = (double *) malloc (N*sizeof(double));
  c[i] = (double *) malloc (N*sizeof(double));
}
...
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        a[i][j] = a[i][j] + c[j][i]*d;
```

```
/* idem em chamada a rotina */

main ()
{
  double a[N][N], c[N][N], d;
  ...
  sub (a,c,d,N)
}

sub (a,c,d,N)
double *a, *c, d;
int n;
{
 ...
}
```

- Medindo o tamanho do programa:

```
% size prog
text    data    bss     dec     hex     filename
4548    172     4       4724    1274    prog
```

- Medindo a atividade de *paging*:

```
% vmstat 5
 procs     memory              page
 r b w   swap  free  re  mf pi po ...
 0 0 0 221784  4632   0   3  3  1 ...
 0 0 0 218160  1536   0   0  0  0 ...
```

EXTENSÕES DE LINGUAGENS COM SUPORTE AO PAD

- Bibliotecas de subrotinas matemáticas (operações com vetores/matrizes, etc.) ou específicas para outro fim:
  - proprietárias, de domínio público ou de terceiros;
  - exemplos: NAG, IMSL, BLAS, LAPACK, etc.
- Pré-processadores vetorizantes, paralelizantes e compiladores com otimização avançada, com diretivas específicas para não-dependências, permutações, não-vetorizar, não-equivalências, probabilidade de desvio, contagem de iterações, "unrolling", não fazer transformações associativas, inlining, níveis de otimização, etc.
- Linguagem explicitamente paralelas *versus* extensões de linguagens para paralelismo;

- **Fortran 90**: operações com vetores e matrizes, alocação dinâmica de memória, ponteiros, novos tipos de dados, estruturas, funções intrínsecas para vetores e matrizes, novas estruturas de controle;
  - operações com vetores e matrizes:

```
DO I = 1, N
   A(I) = A(I) + B(I)
ENDDO

DO J = 1, M
   DO I = 1, N
      A(I,J) = A(I,J) + B(I,J)
   ENDO
ENDDO
```

    * Soma: $\mathbf{A} = \mathbf{A} + \mathbf{B}$
    * Multiplicação elemento a elemento $\mathbf{A} = \mathbf{A} * \mathbf{B}$
    * Multiplicação de matrizes: MATMUL
    * Transposta: TRANSPOSE
    * *shape conformance*
    * *triplets*: exemplo.

```
REAL X(10,10), Y(100), Z(5,5,5)
...
X (10, 1:10)  = Y (91:100)
X (10, :)     = Y (91:100)
Z (10:1:-2,5) = Y (1:5)
```

– funções Intrínsecas:

```
REAL A(100,10,2)
REAL B(10,10,100)
...
A = SIN(A)
B(:, :, 1) = COS (A(1:100:10, :, 1))

C Novas funções intrínsecas:

MAXVAL, MINVAL, DOT_PRODUCT, RESHAPE,
SPREAD, MERGE, SHAPE, SIZE, LBOUND,
UBOUND, ANY, ALL
```

– novas estruturas de controle:

```
C estrutura DO-WHILE;
```

```
C estrutura WHERE-ELSEWHERE-ENDWHERE:
REAL A(4,4), B(4,4), C(4,4)
...
WHERE (B.EQ.C)
      A = 1.0
      C = B + 1.0
ELSEWHERE
      A = -1.0
ENDWHERE

C DO-ENDDO, CYCLE & EXIT:

      I = 0
LOOP: DO
        I = I + 1
        IF (I.EQ.2*(I/2)) CYCLE
        IF (I.GT.10) EXIT
        WRITE(*,*) I
      ENDDO LOOP

C estrutura SELECT-CASE;
```

- **High Performance FORTRAN (HPF)**: extensão do Fortran 90 para máquinas paralelas (comandos extras para distribuir e sincronizar tarefas entreprocessadores);
- Ambientes de programação paralela explícita (exemplos de *message passing languages*: PVM, Linda - baseado em espaço de tuplas);

BENCHMARKING

- **Benchmarks** (programas para medidas de desempenho); tipos (CPU, IO, gráficos, etc);
- **VAX MIPS**: baseado no DEC VAX 11/780 (470 k instruções ponto-fixo por segundo);
- problemas: instruções diferentes gastam tempos diferentes; e o desempenho de acesso à memória?
- MFLOPS (megaflops) - ponto flutuante, aprox. workstations: mflop = clock * 2

- **Linpack**: eliminação gaussiana 100x100 (Linflops), dependente de *daxpy*'s (razão 2:3); cada fabricante pode escolher compilador e opções de compilação;
- **Whetstone**: benchmark de ponto flutuante voltado para funções transcendentais e não para vetorização;
- **SPEC benchmarks**:
  - tal como as anteriores, avalia o desempenho da CPU;
  - **SPECint95** (ponto fixo) e **SPECfp95** (ponto flutuante);
  - média geométrica dos tempos de execução de um conjunto de 5 programas em fp (precisão simples), 9 em fp (precisão dupla) e 6 em ponto fixo;
  - treinador de redes neurais, *compress* do UNIX, método de Monte Carlo, interpretador de LISP, ray tracing,

compilador C da GNU, simulações em química quântica, diferenças finitas para previsão de tempo, equações de Maxwell, planilha de cálculo, etc. (programa p/ desempenho de acesso à memória, removido devido à evolução dos compiladores);

- **Transaction processing benchmarks** (**TPC-A**, **TPC-B**, **TPC-C**), para aplicações interativas com uso de emulador de terminal remoto (automação bancária, etc.);

FAÇA SEU PRÓPRIO BENCHMARKING

- escolha criteriosa de um conjunto de programas que seja representativo;
- run time adequado: não muito curto ou longo;
- comparação da máquina candidata com a atual (tempos *versus* MIPS, etc.);

- tamanhos de dados diferentes podem levar a testar o desempenho da CPU ou do sistema de memória;
- elaboração de *kernel* do software utilizado para benchmarking (quando adequado);
- cuidados com software de terceiros;
- tipos de benchmarking:
  - "**fluxo simples**": conjunto de programas executados sequencialmente, *elapsed time* total e parcial, este normalizado e ponderado (exemplo da figura);
  - "**throughput**": vários programas executados repetida e concorrentemente (A-A-A-A-... com B-B-B-B-... com C-C-C-C-...);
  - **interativos**: uso de emulador de terminal remoto ou não; outros exemplos: processadores de texto, editores, etc.

- Preparando o código para o benchmarking:
  - portabilidade: mínimo de modificações e análise da diferença nos resultados;
  - evitar paging/swapping (verificar elapsed *versus* CPU time);
  - opção mais adequada de otimização de código por compilador;
  - construindo um "benchmark kit" (Makefile ou script, fontes, dados, informação);
  - cuidado com resultados fabulosos do fabricante para seu benchmark;
- Consulte benchmark oficiais também:
  - outros benchmarks da SPEC: **SPECint_rate95**, **SPECfp_rate95**, **SPECweb96**;
  - **http://www.specbench.org**