CAP-387(2016) – Tópicos Especiais em Computação Aplicada: Construção de Aplicações Massivamente Paralelas

Aula 13: Vetorização

Celso L. Mendes, Stephan Stephany

LAC / INPE

Emails: celso.mendes@inpe.br, stephan.stephany@inpe.br

Vetorização

- Princípio Básico:
  - Processar vários operandos através de uma única instrução
- Caso Típico

  C(1)=A(1)+B(1)
  C(2)=A(2)+B(2)
  . . .
  C(N)=A(N)+B(N)

  ou C(1:N)=A(1:N)+B(1:N)

  - Execução: *N* iterações do loop, *6* instruções por iteração: *6.N* instruções
    - Cada instrução precisa passar por busca, decodificação, etc.

Vetorização (cont.)

- Implementação com Instruções Vetoriais:

  loadv A(base) → V1

  loadv B(base) → V2

  addv V1+V2 → V3

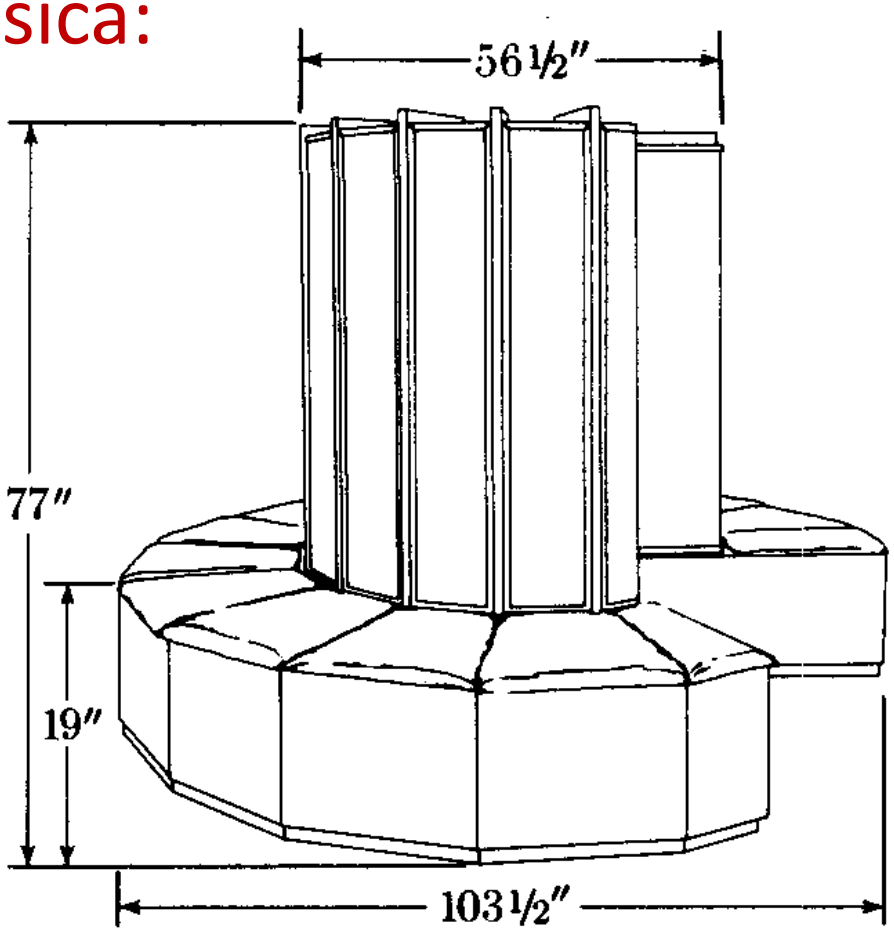
  storev V3 → C(base)

  - V1, V2, V3, etc: registros vetoriais, de comprimento *L*
  - Unidades aritméticas capazes de operar sobre V1, V2, etc
  - Novo total de instruções: 4
  - Se *N* for maior que *L*, montar um laço em torno das instruções acima, com *N/L* iterações

Pioneirismo em Vetorização: Cray-1

Estrutura Física:

77” ≈ 1.95m



Sistema Cray-1: 1976

Diagrama de Blocos:

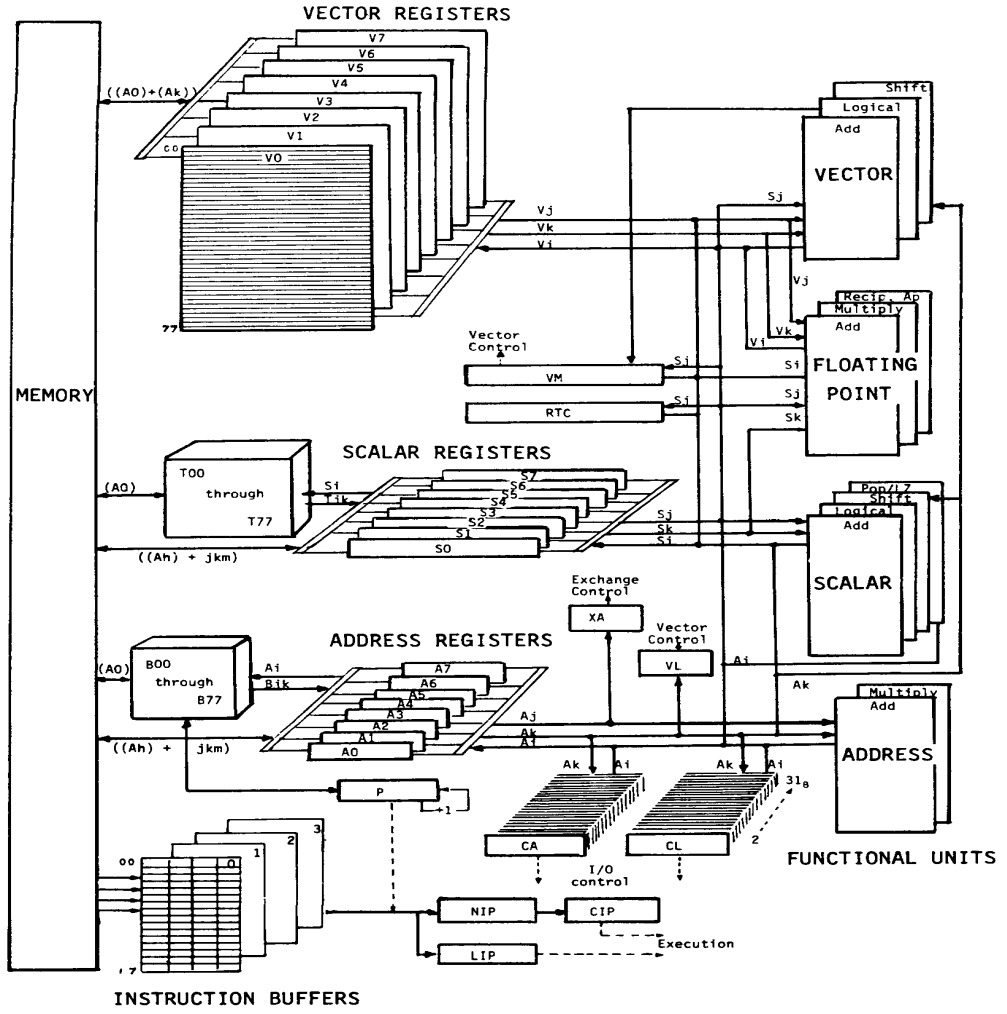
Processamento vetorial →

Processamento escalar →

Processador “rápido”: 80 MHz

Expectativa de vendas: 10

Vendas Reais ≈ 80 !



Cray-1 – Aspectos Interessantes

- Memória:
  - Não havia caches nem memória virtual
- Registros especiais
  - VL: Vector Length register – comprimento ≤ 64
  - VM: Vector Mask register – 64 bits, 1 por elemento
- Unidades funcionais vetoriais de ponto-flutuante:
  - Soma e multiplicação, com pipelining
  - 2 resultados por ciclo: 160 Mflops/s

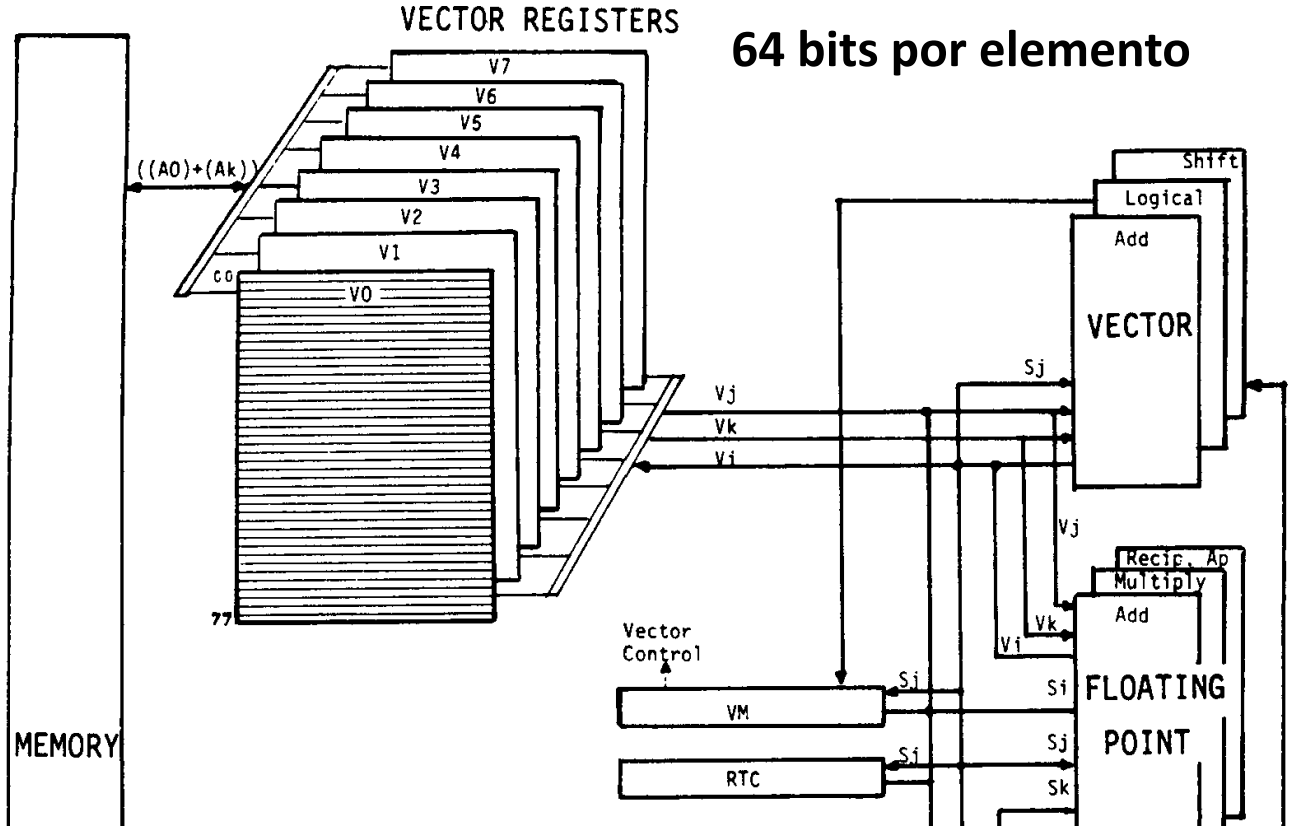
Vetorização (cont.)

- Exemplo: Cray-1

8 Registros Vetoriais (V0,V1,…,V7)

L=64 Elementos em cada registro

64 bits por elemento



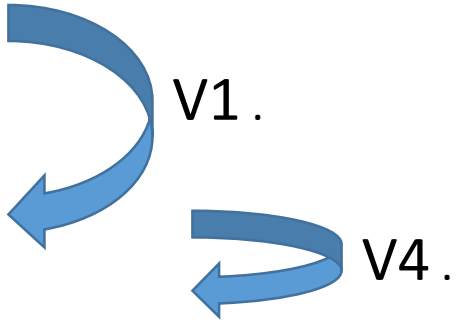
Unidades Funcionais:

Operações Vetor-Vetor ou Vetor-Escalar

Vetorização (cont.)

- Otimizações: encadeamento (*chaining*)
  - Exemplo: D(1:N)=A(1:N)+B(1:N)+C(1:N)
  - Implementação

    loadv A(base) → V1
    loadv B(base) → V2
    loadv C(base) → V3
    addv V1+V2 → V4
    addv V4+V3 → V5
    storev V5 → D(base)



  - Não é necessário esperar o término de toda a primeira operação com Vn para iniciar a segunda operação com Vn (via pipelining)
    - Assim que o primeiro elemento de V1+V2 for produzido, a segunda soma pode ser iniciada! (resultados vão para V4 e para a outra unidade functional)

Vetorização (cont.)

- Otimizações: outros usos de encadeamento
  - Exemplo: Uso de vetores maiores que o tamanho L dos registros vetoriais

```
do K=1,M
        INF = L*(K-1)+1
        SUP = L*K
        C(INF:SUP)=A(INF,SUP)+B(INF,SUP)
enddo
```

Equivalente a

```
C(1:L)=A(1:L)+B(1:L)
C(L+1:2*L)=A(L+1,2*L)+B(L+1,2*L)
C(2*L+1:3*L)=A(2*L+1,3*L)+B(2*L+1,3*L)
. . .
```

  - **Possível implementação**:

```
loadv A(baseA) → V1
loadv B(baseB) → V2
addv  V1+V2 → V3
storev V3 → C(baseC)
loadv A(baseA+N) → V1
loadv B(baseB+N) → V2
addv  V1+V2 → V3
storev V3 → C(baseC+N)
. . .
```

Assim que o primeiro segmento de A tiver sido todo carregado da memória, pode ser iniciada a carga do segundo segmento de A. Idem para B, etc.

Ganho: Unidade de soma sempre gera um novo resultado por ciclo!

Programação para Vetorização

- Suporte em Linguagens
  - Fortran-90 introduziu construções com “notação vetorial” em arrays: código mais claro, vetorização facilitada

    Exemplo: C = A + B

    onde A, B e C são arrays, com mesma dimensão (1D, 2D, …) e tamanho

    No caso de arrays 1D, isto é equivalente a

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

    Exemplo-2: B(1:N)=2*A(4:N+3) (seções de arrays)

    Exemplo-3: B(1:N:2)=2*A(1:N:2) (stride=2)

Compilação para Vetorização

- Compiladores Vetorizadores
  - Geram código vetorizado sempre que possível
  - Emitem relatório de laços onde não é possível vetorizar
    - Causa mais frequente é a *dependência* entre iterações
    - Alternativa: mudar estrutura do laço
  - Amadurecimento dos compiladores foi fundamental para o sucesso dos sistemas vetoriais
  - Sem vetorização, é impossível chegar perto do desempenho de pico do sistema (desempenho obtido fica uma ordem de grandeza abaixo do pico, ou pior)

Restrições à Vetorização

- Possíveis Complicações:

Exemplo: do i=2,N
A(i)=A(i-1)+B(i)
enddo

Implementação "descuidada" com vetorização:

base=2
loadv A(base-1) → V1 A: {10,20,30,40,50,60}
loadv B(base) → V2 B: {11,21,31,41,51,61}
addv V1+V2 → V3 V3: {31,51,71,91,111}
storev V3 → A(base) A:{10,31,51,71,91,111}

Resultado será errado → Valores de A não foram atualizados !

- Um bom compilador não irá vetorizar este loop

Restrições à Vetorização

- Causa do Problema: Dependência de Dados

Exemplo:

```
do i=2,N
      A(i)=A(i-1)+B(i)
enddo
```

```
A(2)=A(1)+B(2)      (i)
A(3)=A(2)+B(3)      (ii)
A(4)=A(3)+B(4)      (iii)
A(5)=A(4)+B(5)      (iv)
. . .
```

Definição: Existe uma **dependência** entre dois commandos quando ambos acessam a *mesma posição de memória*, tal como (i) e (ii), (ii) e (iii), etc.

Tipos de Dependências

- Dependência de Fluxo: Read-After-Write (RAW)

A = B + C
X = A + Y

```
DO i=2,N
        a(i) = a(i-1) + 3
ENDDO
```

- Anti-Dependência: Write-After-Read (WAR)

A = B + C
C = X + Y

```
DO i=1,N-1
        a(i) = a(i+1) + 3
ENDDO
```

- Dependência de Saída: Write-After-Write (WAW)

A = B + C
A = X + Y

```
DO i=1,N
        S = random(i)
ENDDO
```

Tipos de Dependências (cont.)

- Apenas as Dependências de Fluxo são Verdadeiras!
  - Anti-Dependências e Dependências de Saída são causadas pelo reuso de dados, e podem ser eliminadas!
- Anti-Dependência: Write-After-Read (WAR)

```
A = B + C
C = X + Y   P = X + Y
```

```
DO i=1,N-1
   a(i) = a(i+1) + 3   b(i) = a(i+1) + 3
ENDDO
a(1:N-1)=b(1:N-1)
```

- Dependência de Saída: Write-After-Write (WAW)

```
A = B + C
A = X + Y   Q = X + Y
```

```
DO i=1,N
   S = random(i)   S(i)=random(i)
ENDDO
S = S(N)
```

Análise de Dependências

- Como verificar se há dependências entre iterações ?
  - Caso trivial:

```
do i=2,N
    a(i) =a(i-1) + 3
enddo
```

```
i=2:  a(2) = a(1) + 3
i=3:  a(3) = a(2) + 3
i=4:  a(4) = a(3) + 3
. . .
```

  - Caso não-trivial:

```
do i=1,N
    a(24*i-4) = a(5*i+15) + 3
enddo
```

→ Há dependência ???

Se, para algum par {j,k} tal que 1≤{j,k}≤N, 24*j-4 é igual a 5*k+15
então há dependência!

Detecção de Dependência

- Testes de Dependência:
  - Testes aplicados automaticamente pelos compiladores
  - Inúmeros testes disponíveis na literatura
  - Em geral, há diretivas para o programador informar ao compilador que não há dependências num certo loop
  - Casos difíceis para o teste pelo compilador:
    - Limites do loop desconhecidos em tempo de compilação
    - Índices desconhecidos em tempo de compilação (ex: A(K(i))
    - Simplificação comum: restringir os índices a combinações lineares dos índices do loop

      Exemplo: A(4*i-3) = A(3*i+2) + …

Detecção de Dependência

- Exemplo de Teste de Dependência:

  Testes simples: Teste do MDC

  - Equação Diophantine: $a_1x_1 + a_2x_2 + \ldots + a_Nx_N = c$
    - Dados $\{a_1, a_2, \ldots, a_N, c\}$ inteiros, encontrar $\{x_1, x_2, \ldots, x_N\}$ inteiros tais que eles satiafaçam a equação acima.

      Notar que pode não existir solução inteira!
    - Solução: Existe solução inteira *se e somente se* G divide c

      onde $G = MDC(a_1, a_2, \ldots, a_N)$
    - Se houver solução, significa que duas iterações do loop têm referências com dependências!

Detecção de Dependência

- Exemplo-1:

```
do i=1,N
    a(2*i) = a(2*i+1) + 3
enddo
```

Equação Diophantine: $2*x_1 = 2*x_2 + 1$, $(x_1 - x_2) = 1/2$

Como x_1 e x_2 devem ser inteiros, não há solução possível!

Teste do MDC: G = MDC(2,-2) = 2, porém 2 **não** divide 1

Logo, não existe solução possível, e assim não há dependência

→ Loop pode ser vetorizado!

Detecção de Dependência

- Exemplo-2:

```
do i=1,N
      a(19*i+3) = …
                …  = a(2*i+21) + …
enddo
```

Equação Diophantine: $19*x_1+3=2*x_2+21$, $19*x_1-2*x_2=18$

Teste do MDC: G = MDC(19,-2) = 1 e 1 divide 18!

Neste caso, $x_1=2$ e $x_2=10$ são soluções:

```
      i=2:     a(41) = …
      i=10:          … = a(41) + …
```

→ Não há dependência se N ≤ 9 : loop **pode** ser vetorizado

→ Há dependência se N > 9 : loop **não** pode ser vetorizado!

Testes de Dependência

- Teste do MDC
  - Ainda é muito usado hoje, devido à sua simplicidade
  - Caso MDC=1 ocorre com frequência na prática, tornando o teste pouco conclusivo
  - Teste pode ser extendido para mais que uma dimensão, porém a forma do espaço de iteração é ignorada
- Testes de dependência em geral
  - Foco de diversos trabalhos de pesquisa no passado
  - São importantes tanto para *vetorização* como para *paralelização*
    - Ambas as técnicas dependem da relação entre iterações de loops!
- Compiladores atuais
  - Muito bons para vetorizar/paralelizar loops automaticamente, porém…
    - Programador pode ter que mudar o programa para viabilizar isso!