Instituto Tecnológico de Aeronáutica

CE-265: Processamento Paralelo

Exercício 05: Vetorização

Bruno Benjamim Bertucci - Turma 23.2

1 Condições para laços vetorizáveis

1.1 Eliminação de dependências

Para que um laço possa ser vetorizado, é necessário que as manipulações de dados feitas nele não tenham certas dependências. Os tipos de dependência possíveis são:

1. Dependência de fluxo: Uma variável é atribuída e subsequentemente utilizada.
2. Anti-dependência: Uma variável é utilizada e subsequentemente atribuída.
3. Dependência de saída: Uma variável é atribuída e subsequentemente atribuída novamente.

No programa *Lacos.c*, encontrou-se duas funções, *Laco1* e *Laco2*, cada qual apresentando laços com dois tipos de dependência distintos, impedindo a vetorização deles. Na primeira função, havia a seguinte série de comandos:

```
x[i]=y[i]+z[i];
d[i]=x[i+1]+1;
```

Observando esse bloco, percebe-se que há uma dependência no segundo comando: a variável *x[i+1]* utiliza o próximo elemento do vetor, o qual deve ter o seu valor original no momento da execução do comando. Porém, se esse laço fosse vetorizado, o primeiro comando alteraria todo o vetor x antes de prosseguir para o segundo, produzindo uma saída incorreta.

Para eliminar essa dependência, basta trocar a ordem dos dois comandos, permitindo que o laço seja vetorizado. Já a função *Laco2* é composta do seguinte código:

```
float t, s;
t=x[0];
for(int i=1; i<n; i++) {
  s=x[i];
  y[i]=(s+t)/2.0f;
  t=s;
}
```

O laço apresenta tanto dependência de fluxo, com a variável s, quando anti-dependência, com a variável t, o que impede a vetorização dele. Ademais, essas variáveis tornam-se desnecessárias caso a atribuição do vetor y seja feita diretamente como a média de *x[i]* e *x[i-1]*. Fazendo essa alteração e eliminando as variáveis s e t, as dependências são eliminadas e o laço pode ser vetorizado. O código das duas funções vetorizadas está no Anexo A.

1.2 Relatório de vetorização

A vetorização de laços é feita automaticamente pelo compilador *gcc* quando a *flag* de otimização -O3 é utilizada. O uso de uma segunda *flag*, denotada por *-fopt-info-vec-optimized*, produz mensagens em tempo de compilação, que formam um relatório de vetorização e indicam quais laços foram vetorizados com sucesso e quantos laços foram vetorizados em cada função. Trechos do relatório de vetorização do programa *Lacos.c* estão expostos abaixo, demonstrando a vetorização dos laços modificados.

```
Lacos.c:25: note: LOOP VECTORIZED.
Lacos.c:24: note: vectorized 1 loops in function.

Lacos.c:33: note: LOOP VECTORIZED.
Lacos.c:32: note: vectorized 1 loops in function.
```

1.3 Saída do programa modificado

Após compilar com sucesso, o programa *Lacos.c* foi executado, produzindo a saída ilustrada abaixo. A saída produzida pelo programa não vetorizado é idêntica, o que verifica a correção da implementação feita.

```
Antes Laco1: x=132.00; y=  0.00; z=  0.00; d=  0.00
Apos  Laco1: x=136.00; y=  0.00; z=161.00; d=163.00

Antes Laco2: x=132.00; y=  0.00;
Apos  Laco2: x=132.00; y= 14.00
```

2 Vetorização da solução estacionária da Equação de Calor

2.1 Implementação do cálculo vetorizado

A próxima tarefa de vetorização realizada foi de realizar uma aproximação da solução da equação de condução de calor, dada por $\nabla^2 T(x, y) = 0$, para uma placa quadrada discretizada na forma de uma matriz. Isso é feito de tal forma que há duas linhas e duas colunas servindo como extremidades, portanto, a dimensão da matriz quadrada é de $n+2$. A extremidade superior tem temperatura constante de 10^oC, enquanto as demais extremidades têm temperaturas constantes de 0^oC.

A cada iteração do procedimento, cada elemento da matriz, com exceção das linhas e colunas nas extremidades, tem sua temperatura atualizada, sendo seu novo valor dado pela média das temperaturas de seus quatro vizinhos acima, abaixo, à esquerda e à direita. O procedimento original realiza essa atualização linha por linha, o que significa que um dado elemento depende do valor já atualizado do elemento anterior da mesma linha para calcular seu novo valor. Essa dependência impede a realização da vetorização dessa operação.

A função que realiza o cálculo foi reescrita para permitir a sua execução de forma vetorizada. Para esse fim, a ordem da atualização dos elementos foi alterada, passando a ser feita pelas diagonais da matriz em vez de ser feita pelas linhas.

A implementação dessa nova ordem demandou que a função fosse expandida para realizar três operações principais: primeiro, foi criada uma matriz auxiliar, de dimensões $(2n+3)x(n+2)$, e em cada linha dela foi copiada uma diagonal da matriz original; segundo, o cálculo de propagação da temperatura foi realizado, iterando sobre cada linha da matriz auxiliar; terceiro, a matriz original foi atualizada com os valores da matriz auxiliar, transferindo as linhas desta para as diagonais daquela.

Em cada um desses três passos, laços consecutivos foram criados para adequarem as operações aos padrões distintos de indiciação que ocorrem nas diagonais anteriores à diagonal principal da matriz original, na própria diagonal principal, e nas diagonais posteriores a ela. O código da função vetorizada está no Anexo B.

2.2 Relatório de vetorização

Novamente, no momento da compilação, foi gerado um relatório de vetorização pelo compilador para que fosse possível verificar se os laços desejados foram vetorizados com sucesso. No caso da implementação feita, foi constatada a vetorização da porção de código que calcula a propagação de calor, que foi dividida em três conjuntos de laços, conforma indica os trechos do relatório de vetorização mostrados abaixo.

```
Calor.c:57: note: LOOP VECTORIZED.
```

```
Calor.c:52: note: LOOP VECTORIZED.

Calor.c:47: note: LOOP VECTORIZED.

Calor.c:32: note: vectorized 3 loops in function.
```

2.3 Teste de eficiência da vetorização

Para avaliar a vantagem de tempo de execução com a implementação vetorizada, um teste foi conduzido, executando o cálculo da propagação de calor tanto com o procedimento original quanto com o procedimento vetorizado, registrando os tempos exigidos para ambos os cálculos e determinando o maior erro relativo entre elementos correspondentes dos resultados dos dois métodos. É esperado um erro de ordem pequena, uma vez que modifica-se a forma como as operações de ponto flutuante são realizadas. Esse teste foi conduzido com matrizes de tamanho 32 e 10.000, estando os resultados respectivos para cada teste ilustrados abaixo.

```
MaxDif=2.431287e-07
tempo escalar =0.000643
tempo vetorial=0.000170

MaxDif=8.455301e-06
tempo escalar =84.855168
tempo vetorial=12.504681
```

Para uma matriz de dimensão 32, o cálculo com a função vetorizada executou $3,78$ vezes mais rápido do que usando a função escalar, enquanto com dimensão 10000, a execução foi $6,79$ vezes mais rápida. Os erros entre os resultados produzidos pelas duas implementações são pequenos, estando dentro dos valores esperados por conta da diferença na ordem das operações com ponto flutuante, o que verifica a correção da implementação vetorizada.

Pode-se notar que os ganhos de velocidade com a implementação vetorizada são bastante expressivos, especialmente para matrizes com tamanhos maiores. Entretanto, a dificuldade da implementação também foi elevada, uma vez que a função que era composta originalmente por três laços aninhados e um único comando dentro deles se tornou uma série extensa de laços, com uma lógica de indexação mais complexa que a original.

Ademais, com a realização dessa tarefa, foi possível perceber que a vetorização pode ser de difícil implementação para alguns problemas, tornando o código mais extenso e menos compreensível. Porém, os ganhos de desempenho são significativos. Portanto, pode-se dizer que a vetorização de tarefas complexas é interessante quando é esperado que o programa em questão seja usado frequentemente com uma quantidade elevada de dados, que necessite de uma precisão elevada, ou que seja usado em situações com restrições significativas de capacidade de processamento ou de tempo de execução.

Anexo A – Código de laços vetorizados

```
void Laco1(float* restrict x, float* restrict y,
     float* restrict z, float* restrict d, int n) {
  for(int i=1; i<n; i++) {
    d[i]=x[i+1]+1;
    x[i]=y[i]+z[i];
    z[i]=d[i];
  }
}

void Laco2(float* restrict x, float* restrict y, int n) {
  for(int i=1; i<n; i++) {
    y[i]=(x[i]+x[i-1])/2.0f;
  }
}
```

Anexo B – Código vetorizado da solução da Equação de Calor

```
// resolve a equacao de calor com codigo vetorial

void CalorVetorial(const int nIter, const int n, float b[n+2][n+2]) {
  float c[2*n+3][n+2];
  int i, j, s, t;
  for(j=0; j<(n+2); ++j) {
    c[n+1][j] = b[n+1][j];
  }
  for (i=0; i<(n+1); ++i) {
    for(j=0; j<=(i+1); ++j) {
      c[i][j] = b[i-j][j];
      c[2*n+2-i][j] = b[n+1-j][n+1-(i-j)];
    }
  }

  for (int iIter=0; iIter<nIter; ++iIter) {
    for (s=2; s<n+1; ++s) {
      for (t=1; t<=s-1; ++t) {
        c[s][t]=0.25*(c[s-1][t-1] + c[s-1][t]+
                      c[s+1][t] + c[s+1][t+1]);
      }
    }
    for (t=1; t<=n; ++t) {
      c[n+1][t]=0.25*(c[n][t-1] + c[n][t]+
        c[n+2][t-1] + c[n+2][t]);
    }
    for (s=n+2; s<=2*n; ++s) {
      for (t=1; t<=(2*n)-(s-1); ++t) {
        c[s][t]=0.25*(c[s-1][t] + c[s-1][t+1]+
                      c[s+1][t-1] + c[s+1][t]);
      }
    }
  }

  for (i=0; i<(n+1); ++i) {
    for(j=0; j<=(i+1); ++j) {
      b[i-j][j] = c[i][j];
    }
  }
  for (i=0; i<(n+1); ++i) {
    for(j=0; j<=(i+1); ++j) {
        b[n+1-j][n+1-(i-j)] = c[2*n+2-i][j];
    }
  }
  for(j=0; j<(n+2); ++j) {
    b[n+1-j][j] = c[n+1][j];
  }
}
```