Nesta atividade foram testados os efeitos do paralelismo ao nível de instrução (ILP) no desempenho da soma dos elementos de um vetor. O objetivo foi comparar diferentes abordagens para a soma, analisando a influência das dependências entre as iterações e explorando técnicas como o uso de múltiplas variáveis para quebrar essas dependências. Além disso, medi o tempo de execução das versões compiladas com diferentes níveis de otimização do gcc (O0, O2 e O3), verificando como o compilador pode influenciar a performance do código.

Conceitos de ILP e Pipeline

O paralelismo ao nível de instrução (ILP – Instruction Level Parallelism) refere-se à capacidade do processador de executar múltiplas instruções simultaneamente, desde que não existam dependências que impeçam essa execução paralela. Para explorar esse paralelismo, os processadores modernos utilizam um conceito chamado pipeline, onde diferentes estágios do processador trabalham em conjunto para processar várias instruções ao mesmo tempo.

Entretanto, um fator que pode limitar a eficiência do pipeline é a presença de dependências de dados. Por exemplo, se uma operação depende do resultado da anterior, o processador precisa esperar a conclusão da primeira antes de executar a segunda, reduzindo o grau de paralelismo.

Implementação

Para analisar esses conceitos, implementei três funções no código:

1. **inicializarVetor**: Preenche um vetor de tamanho N = 100000000 com valores gerados a partir de uma equação simples:

$$v[i] = i * 2 + 1$$

2. **somaAcumulativa**: Percorre o vetor e soma seus elementos de maneira sequencial, criando uma dependência entre as operações.
3. **somaPipeline**: Busca explorar o paralelismo ao quebrar essa dependência. Utiliza-se quatro variáveis acumuladoras, permitindo que diferentes partes do vetor sejam somadas em paralelo. Esse método reduz a dependência entre operações consecutivas, aumentando a eficiência do pipeline do processador.

Cada função foi executada e teve seu tempo medido com três diferentes níveis de otimização:

- **O0**: Nenhuma otimização, mantendo o código próximo ao escrito.
- **O2**: Nível alto de otimização, tentando reduzir instruções redundantes e melhorar o desempenho sem comprometer a precisão.
- **O3**: Nível máximo de otimização, onde o compilador faz otimizações mais agressivas, incluindo vetorização e reordenação de instruções.

Resultados

Os tempos obtidos foram os seguintes:

Otimização	somaAcumulativa	somaPipeline
O0	0.236000	0.081000
O2	0.074000	0.063000
O3	0.080000	0.063000

Podemos observar que:

1. Sem otimização O0, a soma acumulativa é significativamente mais lenta do que a soma com pipeline. Isso acontece porque o código original não aproveita bem o pipeline do processador, enquanto a versão otimizada quebra dependências e permite que mais operações ocorram simultaneamente.
2. Com otimização O2, há uma grande redução no tempo de execução. O compilador realiza otimizações automáticas, como reorganização de instruções e eliminação de operações desnecessárias. Nesse caso, a diferença entre as duas abordagens diminui, pois o compilador já aplica algumas técnicas de ILP automaticamente.
3. Com otimização O3, o tempo da soma acumulativa aumenta levemente em relação ao O2, o que pode ser devido a alguma reordenação interna do código pelo compilador. Já a soma com pipeline mantém o mesmo tempo do O2, indicando que já estava suficientemente otimizada.

Esses resultados mostram como a dependência de dados impacta o desempenho e como técnicas como pipeline podem melhorar a eficiência do código. Além disso, evidenciam que o compilador pode aplicar otimizações automaticamente, reduzindo a necessidade de modificações manuais no código, dependendo do nível de otimização escolhido.

Código Utilizado:

tarefa2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 100000000

void inicializarVetor(double v[N]) {
    for (int i = 0; i < N; i++) {
        v[i] = i * 2.0 + 1.0;
    }
}

double somaAcumulativa(double v[N]) {
    double soma = 0.0;
    for (int i = 0; i < N; i++) {
        soma += v[i];
    }
    return soma;
}

double somaPipeline(double v[N]) {
    double s1 = 0.0, s2 = 0.0, s3 = 0.0, s4 = 0.0;

    for (int i = 0; i < N; i += 4) {
        s1 += v[i];
        s2 += v[i + 1];
        s3 += v[i + 2];
        s4 += v[i + 3];
    }

    return s1 + s2 + s3 + s4;
}
```

```c

int main() {
    static double vetor[N];

    inicializarVetor(vetor);

    clock_t inicio = clock();
    double resultado1 = somaAcumulativa(vetor);
    clock_t fim = clock();
    double tempo1 = (double)(fim - inicio) / CLOCKS_PER_SEC;

    inicio = clock();
    double resultado2 = somaPipeline(vetor);
    fim = clock();
    double tempo2 = (double)(fim - inicio) / CLOCKS_PER_SEC;

    printf("Soma acumulativa: %f (Tempo: %f segundos)\n", resultado1, tempo1);
    printf("Soma com pipeline: %f (Tempo: %f segundos)\n", resultado2, tempo2);

    return 0;
}
```

Saída do código: