UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

Relatório da Tarefa 02 - Os efeitos do paralelismo ao nível de instrução (ILP)
DCA3703 - PROGRAMAÇÃO PARALELA - T01 (2025.2)

WERBERT ARLES DE SOUZA BARRADAS

20250070655

Docente: Professor Doutor SAMUEL XAVIER DE SOUZA

Natal, 20 de agosto de 2025

Lista de Figuras

Sumário

1 Introdução

As arquiteturas de computadores modernas dependem fundamentalmente do **Paralelismo em Nível de Instrução (ILP - Instruction-Level Parallelism)** para alcançar alto desempenho. Processadores superescalares são projetados para executar múltiplas instruções por ciclo de clock, utilizando diversas unidades funcionais (ALUs, FPUs) em paralelo. No entanto, a capacidade de explorar o ILP é severamente limitada por dependências de dados no código, que serializam a execução e subutilizam os recursos do hardware.

Este estudo tem como objetivo investigar e quantificar o impacto das dependências de dados na exploração do ILP. Para isso, foram implementados três laços computacionais em C: um laço de inicialização de vetor, um laço de soma sequencial com forte dependência de dados, e um terceiro laço que quebra essa dependência através da técnica de *loop unrolling* e uso de múltiplos acumuladores.

O desempenho de cada laço foi medido sob diferentes níveis de otimização do compilador GCC (`-O0`, `-O2`, `-O3`). A análise dos resultados permite elucidar como a estrutura do código-fonte interage com a microarquitetura do processador e como as otimizações do compilador são cruciais para expor e explorar o paralelismo inerente ao hardware.

2 Metodologia do Experimento

Para analisar o ILP, foi desenvolvido um programa em C que executa três operações distintas sobre um vetor de 100 milhões de inteiros. Cada operação foi encapsulada em um laço, projetado para exibir diferentes características de dependência de dados.

2.1 Laço 1: Inicialização do Vetor

O primeiro laço é responsável por popular o vetor com valores simples. A operação em cada iteração é `vetor[i] = (i % 10) + 1;`.

```
for (int i = 0; i < TAMANHO_VETOR; i++) {
    vetor[i] = (i % 10) + 1;
}
```

Listing 1 – Laço 1: Inicialização do vetor.

Do ponto de vista da arquitetura, este laço é altamente paralelizável. Cada iteração é **completamente independente** das outras. Uma escrita em `vetor[i]` não afeta a escrita em `vetor[i+1]`. Processadores modernos com execução *out-of-order* podem processar múltiplas iterações simultaneamente, utilizando técnicas como *pipelining* e múltiplas unidades de armazenamento para esconder a latência da memória.

2.2 Laço 2: Soma com Dependência de Dados

O segundo laço calcula a soma de todos os elementos do vetor usando um único acumulador.

```
long long soma_dependente = 0;
for (int i = 0; i < TAMANHO_VETOR; i++) {
    soma_dependente += vetor[i];
}
```

Listing 2 – Laço 2: Soma com dependência de dados (Read-After-Write).

Este código cria uma **dependência de dados verdadeira (Read-After-Write - RAW)** na variável `soma_dependente`. A iteração `i+1` só pode ler o valor de `soma_dependente` depois que a iteração `i` o escreveu. Isso cria uma longa cadeia de dependência que serializa a execução.

O pipeline do processador é forçado a parar (*stall*) a cada iteração, aguardando o resultado da adição anterior. Mesmo em uma arquitetura superescalar com múltiplas ALUs, apenas uma pode ser utilizada para esta tarefa, tornando a exploração do ILP praticamente impossível.

2.3 Laço 3: Quebra de Dependência

O terceiro laço implementa a mesma soma, mas quebra a cadeia de dependência utilizando múltiplos acumuladores. A técnica, conhecida como *loop unrolling*, processa múltiplos elementos do vetor por iteração.

```
long long s1=0, s2=0, s3=0, s4=0;
for (int i = 0; i < TAMANHO_VETOR; i += 4) {
    s1 += vetor[i];
    s2 += vetor[i+1];
    s3 += vetor[i+2];
    s4 += vetor[i+3];
}
long long soma_total_indep4 = s1 + s2 + s3 + s4;
```

Listing 3 – Laço 3: Soma com quebra de dependência (fator 4).

2.4 Laço 4: Maximizando o ILP (Fator 8)

O quarto laço estende a técnica anterior, utilizando oito acumuladores. O objetivo é tentar saturar as unidades de execução do processador, fornecendo um grau ainda maior de paralelismo de instrução para o hardware explorar.

```
long long a1=0, a2=0, a3=0, a4=0, a5=0, a6=0, a7=0, a8=0;
for (int i = 0; i < TAMANHO_VETOR; i += 8) {
    a1 += vetor[i];   a2 += vetor[i+1]; a3 += vetor[i+2]; a4 +=
    ↪  vetor[i+3];
    a5 += vetor[i+4]; a6 += vetor[i+5]; a7 += vetor[i+6]; a8 +=
    ↪  vetor[i+7];
}
long long soma_total_indep8 = a1+a2+a3+a4+a5+a6+a7+a8;
```

Listing 4 – Laço 4: Soma com quebra de dependência (fator 8).

Arquitetonicamente, esta é a abordagem mais eficiente. As quatro operações de soma dentro do laço são **independentes entre si**. O processador pode despachar cada uma das quatro adições (`s1 += ...`, `s2 += ...`, etc.) para diferentes unidades de

execução (ALUs) para serem executadas **em paralelo no mesmo ciclo de clock**. Isso explora diretamente o ILP do hardware. A cadeia de dependência agora é quatro vezes menor, pois cada acumulador só depende de si mesmo a cada quatro elementos.

O Laço 4 estende essa mesma lógica ao limite, utilizando oito acumuladores. O objetivo é fornecer um grau de paralelismo de instrução ainda maior, na tentativa de saturar completamente as unidades de execução do processador. Com oito somas independentes disponíveis por iteração, o agendador de instruções do processador tem a máxima flexibilidade para manter múltiplas ALUs ocupadas a cada ciclo.

No entanto, esta abordagem encontra a lei dos **rendimentos decrescentes**. Embora o ganho de desempenho de 1 para 4 acumuladores seja massivo, o ganho de 4 para 8 é incremental e visivelmente menor. Isso ocorre porque o gargalo da performance deixa de ser a dependência de dados no código e passa a ser os próprios limites físicos do hardware: o número de ALUs disponíveis, a largura de banda do pipeline para buscar e decodificar instruções, ou a quantidade de registradores físicos para renomeação. O código agora oferece mais paralelismo do que a microarquitetura consegue executar simultaneamente, aproximando-se do desempenho máximo teórico para esta tarefa.

2.5 Procedimentos de Teste

O programa foi compilado utilizando o GCC (GNU Compiler Collection) com três flags de otimização distintas:

- `-O0`: Nenhuma otimização. O código é compilado de forma literal.
- `-O2`: Otimizações moderadas, que não aumentam o tamanho do código. Inclui agendamento de instruções e outras técnicas para melhorar o desempenho.
- `-O3`: Otimizações agressivas, incluindo vetorização e *loop unrolling* automático, que podem aumentar o tamanho do código.

O tempo de execução de cada laço foi medido utilizando `clock_gettime(CLOCK_MONOTONIC, ...)` para alta precisão. Os resultados foram salvos em um arquivo CSV para análise posterior.

3 Resultados

A execução do programa sob os diferentes níveis de otimização produziu os resultados consolidados na Tabela 1.

Tabela 1 – Tempos de execução (em segundos) por laço e nível de otimização.

Tipo de Laço	**-O0**	**-O2**	**-O3**
Laço 1 (Inicialização)	0.494s	0.110s	0.120s
Laço 2 (Com Dependência)	0.220s	0.044s	0.051s
Laço 3 (Fator 4)	0.104s	0.047s	0.047s
Laço 4 (Fator 8)	0.077s	0.038s	0.047s

Os dados da Tabela 1 são visualizados no gráfico de barras da Figura 2.

4 Gráficos

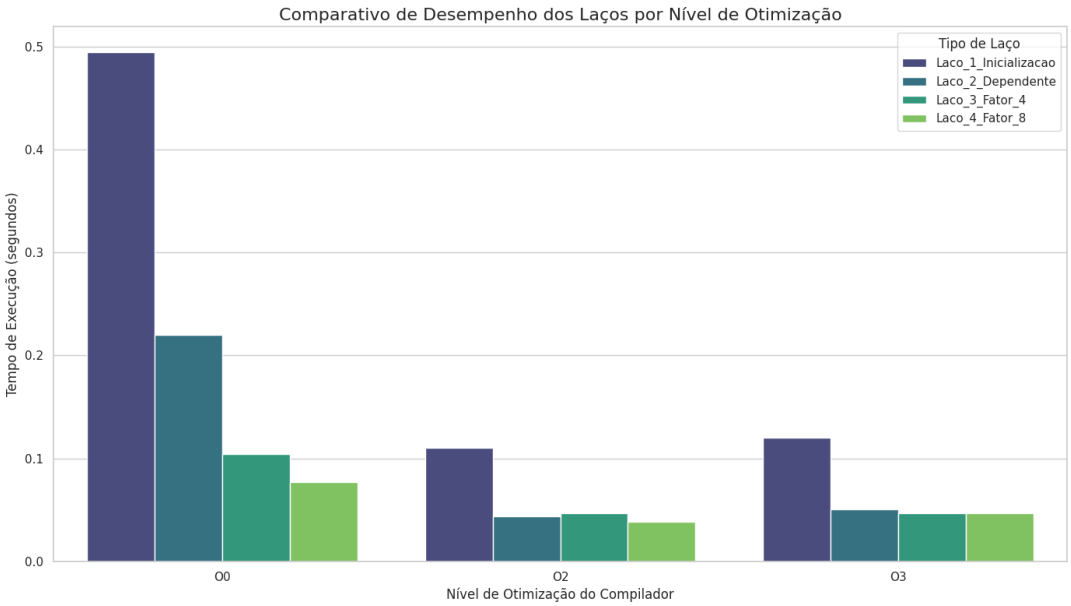

Figura 2 – Comparativo de desempenho entre os laços com diferentes níveis de otimização.

5 Conclusão

A análise dos dados experimentais revela uma complexa interação entre o estilo de codificação, as otimizações do compilador e a microarquitetura do processador.

Análise com -O0 (Baseline): Sem qualquer otimização, os resultados alinham-se perfeitamente com a teoria. O Laço 2 (0.220s), com sua dependência sequencial, é significativamente mais lento que o Laço 1 (0.494s), que é limitado por operações de memória. A quebra manual de dependências mostra seu valor: o Laço 3 (0.104s) é duas vezes mais rápido que o Laço 2, e o Laço 4 (0.077s) é quase três vezes mais rápido. Isso demonstra que, mesmo sem a ajuda do compilador, a arquitetura *out-of-order* do processador consegue explorar o ILP explícito fornecido pelo código.

Impacto das Otimizações (-O2 e -O3): A ativação das otimizações do compilador altera drasticamente o cenário. Com `-O2`, todos os laços apresentam ganhos massivos. O Laço 2, por exemplo, acelera 5 vezes (de 0.220s para 0.044s), indicando que o compilador consegue reorganizar o código para mitigar parcialmente a dependência. O Laço 4 continua a ser o mais rápido, com 0.038s, mostrando que a combinação de código ILP-amigável e otimização do compilador produz o melhor resultado.

O resultado mais intrigante surge com a otimização `-O3`. O desempenho dos Laços 3 e 4 converge para um valor idêntico de 0.047s. Este fenómeno sugere fortemente que o compilador ativou a **vetorização SIMD (Single Instruction, Multiple Data)**. Com `-O3`, o compilador é inteligente o suficiente para reconhecer o padrão de soma e transformá-lo em instruções vetoriais (como SSE ou AVX), que processam múltiplos elementos de dados com uma única instrução. Uma vez que o código é vetorizado, o fator de *unrolling* manual (4 ou 8) torna-se irrelevante, pois o gargalo de desempenho desloca-se da dependência de dados para a largura de banda das unidades de execução SIMD e da memória. O compilador gerou, para ambos os casos, um código final otimizado que satura os recursos de hardware disponíveis.

Conclusão Final: Este experimento valida que a performance não reside apenas no algoritmo, mas na sua expressão em código e na subsequente tradução pelo compilador para a linguagem da máquina. A lição fundamental é dupla:

1. Escrever código que expõe o paralelismo, como a quebra de dependências de dados, é crucial. Sem isso, como visto no Laço 2, nem o compilador mais avançado consegue extrair o máximo de desempenho.

2. Confiar em compiladores modernos é igualmente importante. Os resultados com `-O3` mostram que o compilador pode aplicar otimizações complexas como a vetorização,

que superam em eficiência as otimizações manuais, atingindo um ponto de saturação de performance ditado pelos limites do hardware.

ilp_grafico.c

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

#define TAMANHO_VETOR 100000000

// Função para registrar os dados no arquivo CSV
void registrar_csv(FILE *arquivo, const char *otimizacao, const char *laco, double tempo) {
    if (arquivo) {
        fprintf(arquivo, "%s;%s;%f\n", otimizacao, laco, tempo);
    }
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Uso: %s <Nivel_Otimizacao>\n", argv[0]);
        return 1;
    }
    char *nivel_otimizacao = argv[1];

    FILE *arquivo_csv = fopen("dados_desempenho.csv", "a");
    if (arquivo_csv == NULL) {
        perror("Erro ao abrir o arquivo CSV");
        return 1;
    }

    int *vetor = (int *)malloc(TAMANHO_VETOR * sizeof(int));
    if (vetor == NULL) {
        fprintf(stderr, "Falha na alocacao de memoria\n");
        fclose(arquivo_csv);
        return 1;
    }

    struct timespec inicio, fim;
    double tempo_gasto;

    // --- Laço 1: Inicialização do Vetor ---
    clock_gettime(CLOCK_MONOTONIC, &inicio);
    for (int i = 0; i < TAMANHO_VETOR; i++) {
        vetor[i] = (i % 10) + 1;
    }
    clock_gettime(CLOCK_MONOTONIC, &fim);
    tempo_gasto = (fim.tv_sec - inicio.tv_sec) + (fim.tv_nsec - inicio.tv_nsec) / 1e9;
    printf("Laço 1 (Inicializacao).................: %f segundos\n", tempo_gasto);
    registrar_csv(arquivo_csv, nivel_otimizacao, "Laco_1_Inicializacao", tempo_gasto);

    // --- Laço 2: Soma Acumulativa ---
    long long soma_dependente = 0;
    clock_gettime(CLOCK_MONOTONIC, &inicio);
    for (int i = 0; i < TAMANHO_VETOR; i++) {
```

```c
        soma_dependente += vetor[i];
    }
    clock_gettime(CLOCK_MONOTONIC, &fim);
    tempo_gasto = (fim.tv_sec - inicio.tv_sec) + (fim.tv_nsec - inicio.tv_nsec) / 1e9;
    printf("Laço 2 (Soma com Dependencia).....: %f segundos\n", tempo_gasto);
    registrar_csv(arquivo_csv, nivel_otimizacao, "Laco_2_Dependente", tempo_gasto);

    // --- Laço 3: Quebra de Dependências (Fator 4) ---
    long long s1=0, s2=0, s3=0, s4=0;
    clock_gettime(CLOCK_MONOTONIC, &inicio);
    for (int i = 0; i < TAMANHO_VETOR; i += 4) {
        s1 += vetor[i]; s2 += vetor[i+1]; s3 += vetor[i+2]; s4 += vetor[i+3];
    }
    long long soma_total_indep4 = s1 + s2 + s3 + s4;
    clock_gettime(CLOCK_MONOTONIC, &fim);
    tempo_gasto = (fim.tv_sec - inicio.tv_sec) + (fim.tv_nsec - inicio.tv_nsec) / 1e9;
    printf("Laço 3 (Quebra de Dependencia, Fator 4): %f segundos\n", tempo_gasto);
    registrar_csv(arquivo_csv, nivel_otimizacao, "Laco_3_Fator_4", tempo_gasto);

    // --- Laço 4: Quebra de Dependências (Fator 8) ---
    long long a1=0, a2=0, a3=0, a4=0, a5=0, a6=0, a7=0, a8=0;
    clock_gettime(CLOCK_MONOTONIC, &inicio);
    for (int i = 0; i < TAMANHO_VETOR; i += 8) {
        a1 += vetor[i];   a2 += vetor[i+1]; a3 += vetor[i+2]; a4 += vetor[i+3];
        a5 += vetor[i+4]; a6 += vetor[i+5]; a7 += vetor[i+6]; a8 += vetor[i+7];
    }
    long long soma_total_indep8 = a1+a2+a3+a4+a5+a6+a7+a8;
    clock_gettime(CLOCK_MONOTONIC, &fim);
    tempo_gasto = (fim.tv_sec - inicio.tv_sec) + (fim.tv_nsec - inicio.tv_nsec) / 1e9;
    printf("Laço 4 (Quebra de Dependencia, Fator 8): %f segundos\n", tempo_gasto);
    registrar_csv(arquivo_csv, nivel_otimizacao, "Laco_4_Fator_8", tempo_gasto);



    //Um resultado final para garantir que todas as somas são essenciais.
    long long total_geral = soma_dependente + soma_total_indep4 + soma_total_indep8;
    printf("\nVerificacao de Somas (Dependente: %lld, Fator 4: %lld, Fator 8: %lld)\n",
           soma_dependente, soma_total_indep4, soma_total_indep8);

    free(vetor);
    fclose(arquivo_csv);

    // O valor de retorno do programa agora depende dos resultados.
    return (int)(total_geral % 256);
}
```