

1 Introdução

Este relatório apresenta os resultados do estudo realizado para analisar os efeitos de **Pipeline** e **Vetorização** em trechos de código em C. O estudo envolveu a implementação e execução de três laços:

1. Inicialização de um vetor com um cálculo simples;
2. Soma acumulativa sequencial, que cria dependência entre as iterações;
3. Soma utilizando múltiplas variáveis para quebrar a dependência.

As execuções do código foram realizadas com diferentes níveis de otimização do compilador (-O0, -O2 e -O3), permitindo avaliar como as técnicas de *loop unrolling*, *vetorização* e reorganização de instruções impactam a performance.

2 Metodologia

Para os testes, foi desenvolvido um programa em C que executa os três laços e mede o tempo de execução utilizando a função `clock()` convertendo os ticks para segundos. O vetor testado possui $N = 10^9$ elementos, garantindo que as diferenças de desempenho sejam evidentes dado seu tamanho.

O procedimento adotado foi:

- **O código:** Implementado como solicitado pela tarefa colocando cada um dos loops em uma função.

Para o loop de inicialização do array temos:

```
void init_array(double *array, int size) {  
    for (int i = 0; i < size; i++) {  
        array[i] = i * 0.3;  
    }  
}
```

Para o loop com soma acumulativa dependente:

```
double sum_cumulative(double *array, int size) {  
    double sum = 0;  
    for (int i = 0; i < size; i++) {  
        sum += array[i];  
    }  
    return sum;  
}
```

Para o loop de soma acumulativa de múltiplas variáveis:

```
double sum_parallel(double *array, int size) {
    double sum1 = 0;
    double sum2 = 0;
    for (int i = 0; i < size; i += 2) {
        sum1 += array[i];
        if (i + 1 < size) {
            sum2 += array[i + 1];
        }
    }
    return sum1 + sum2;
}
```

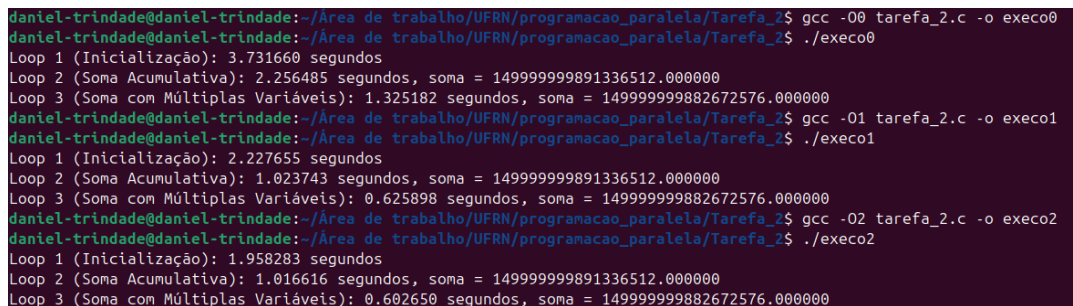
- **Compilação:** O código foi compilado utilizando as diretivas `-O0`, `-O2` e `-O3`:

```
gcc -O0 tarefa_2.c -o exec_00
gcc -O2 tarefa_2.c -o exec_02
gcc -O3 tarefa_2.c -o exec_03
```

- **Medição:** Para cada versão compilada, foram registrados os tempos de execução dos laços:
 - **Loop 1:** Inicialização do vetor, sem dependências, que permite uma boa aplicação de vetorização.
 - **Loop 2:** Soma acumulativa, onde a dependência entre iterações impede o aproveitamento completo do pipeline.
 - **Loop 3:** Soma com múltiplas variáveis, técnica que quebra a dependência e permite otimizações como *loop unrolling*.
- **Análise:** Foram comparados os tempos de execução entre as diferentes diretivas para evidenciar os ganhos de desempenho proporcionados pelas otimizações.

3 Resultados e Discussão

Como resultado obtivemos o tempo de execução para cada laço e como esse tempo muda a medida que trocamos o nível de otimização utilizada para compilar. Na figura a baixo podemos ver o retorno do código:



```
daniel-trindade@daniel-trindade:~/Área de trabalho/UFRN/programacao_paralela/Tarefa_2$ gcc -O0 tarefa_2.c -o execo0
daniel-trindade@daniel-trindade:~/Área de trabalho/UFRN/programacao_paralela/Tarefa_2$ ./execo0
Loop 1 (Inicialização): 3.731660 segundos
Loop 2 (Soma Acumulativa): 2.256485 segundos, soma = 149999999891336512.000000
Loop 3 (Soma com Múltiplas Variáveis): 1.325182 segundos, soma = 149999999882672576.000000
daniel-trindade@daniel-trindade:~/Área de trabalho/UFRN/programacao_paralela/Tarefa_2$ gcc -O1 tarefa_2.c -o execo1
daniel-trindade@daniel-trindade:~/Área de trabalho/UFRN/programacao_paralela/Tarefa_2$ ./execo1
Loop 1 (Inicialização): 2.227655 segundos
Loop 2 (Soma Acumulativa): 1.023743 segundos, soma = 149999999891336512.000000
Loop 3 (Soma com Múltiplas Variáveis): 0.625898 segundos, soma = 149999999882672576.000000
daniel-trindade@daniel-trindade:~/Área de trabalho/UFRN/programacao_paralela/Tarefa_2$ gcc -O2 tarefa_2.c -o execo2
daniel-trindade@daniel-trindade:~/Área de trabalho/UFRN/programacao_paralela/Tarefa_2$ ./execo2
Loop 1 (Inicialização): 1.958283 segundos
Loop 2 (Soma Acumulativa): 1.016616 segundos, soma = 149999999891336512.000000
Loop 3 (Soma com Múltiplas Variáveis): 0.602650 segundos, soma = 149999999882672576.000000
```

Figure 1: Resultados da execução do código com níveis diferentes de otimização

A seguir organizamos os resultados obtidos em uma tabela com os níveis de otimização nas colunas e os loops trabalhados nas linhas facilitando assim a análise:

	-O0	-O2	-O3
Loop 1	3.731660s	2.227655s	1.958283s
Loop 2	2.256485s	1.023743s	1.016616s
Loop 3	1.325182s	0.625898s	0.602650s

Table 1: Tempo de execução (em segundos) para diferentes níveis de otimização

Se considerarmos o o0 como ponto de partida para sabermos o ganho de tempo que temos em cada otimização teremos o seguinte gráfico:

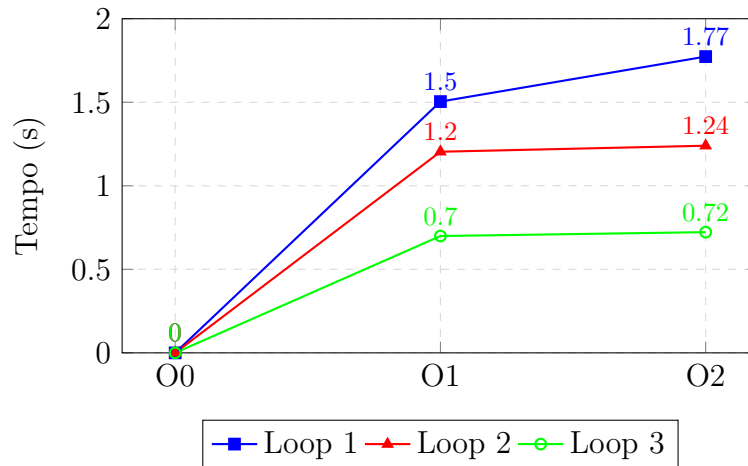


Figure 2: Comparação do Tempo de Execução por Nível de Otimização

Analisando os resultados dos loops individualmente a medida que aplicamos as otimizações, podemos observar que:

- **Loop 1:** Apresenta o melhor ganho de desempenho ao aplicarmos as otimizações. Isso se deve ao fato de que o loop 1 é um candidato ideal para vetorização, porque ele tem operações simples e previsíveis, não há dependências entre iterações e a CPU pode carregar e armazenar múltiplos valores simultaneamente usando registradores vetoriais;
- **Loop 2:** O desempenho é inferior devido à soma sequencial, em que cada iteração depende da anterior, limitando assim as otimizações, por exemplo o Loop 2 requer mais acessos simultâneos à memória, o que pode causar mais "cache misses" (falhas na cache), aumentando a latência. Além disso, como estamos somando valores ao próprio vetor, o processador precisa esperar o valor anterior estar carregado antes de prosseguir, limitando a otimização. Ainda assim ele tem um certo ganho de desempenho com as otimizações pois o compilador consegue aplicar parcialmente a vetorização fazendo com que algumas operações possam ser realizadas em paralelo;
- **Loop 3:** Teve o menor ganho de desempenho em relação aos outros loops quando aumentamos o nível de otimização porque o Loop 3 já reduz dependências de dados em seu código. Esse loop utiliza duas variáveis (sum1 e sum2) para acumular valores de elementos intercalados do array. Isso reduz a dependência entre iterações consecutivas do laço, permitindo que o compilador já aproveite paralelismo e vetorização, mesmo sem otimizações agressivas. Ou seja, o Loop 3 já nasce otimizado. Com isso, os ganhos adicionais das otimizações -O1 e -O2 são limitados.

As diretivas de compilação influenciam fortemente esses resultados pois são elas que controlam o nível de otimização que será aplicada da seguinte forma:

- -O0 não aplica otimizações, resultando em execução mais lenta e nas operações ocorrendo na ordem exata do código.
- -O2 já emprega otimizações moderadas, como vetorização e inlining, melhorando o desempenho dos laços sem dependências.
- -O3 utiliza otimizações agressivas, maximizando o ganho de performance, principalmente em laços que permitem reordenação das instruções.

4 Conclusão

Os experimentos realizados evidenciam a importância de se estruturar o código de forma a favorecer a extração de paralelismo a nível de instrução (ILP). A quebra de dependência, conforme demonstrado no Loop 1, permite que o compilador aplique técnicas avançadas de otimização, resultando em ganhos expressivos de desempenho. Este estudo reforça a relevância das otimizações de *pipeline* e *vetorização* no desenvolvimento de aplicações em computação paralela.