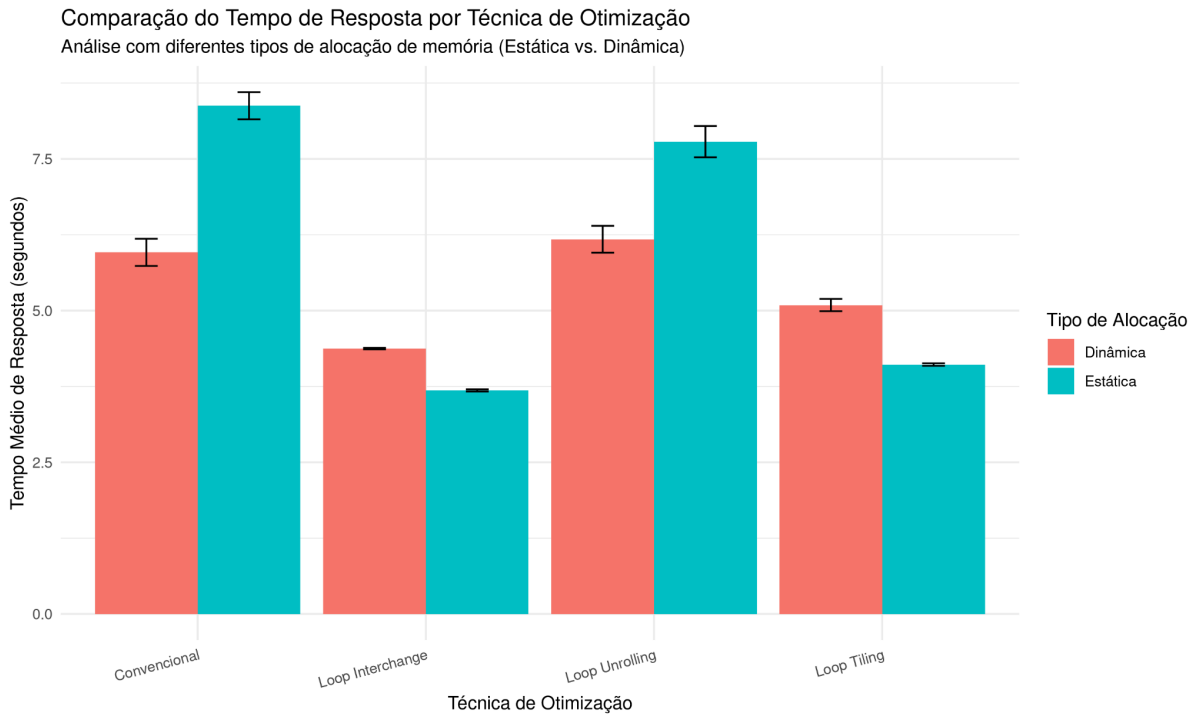
Artur Kenzo Obara Kawazoe - 15652663
Daniel Jorge Manzano - 15446861
Gabriel Phelippe Prado - 15453730
Jhonatan Barboza da Silva - 15645049

Atividade 2
Desenvolvimento de Código Otimizado

Neste relatório, estudamos técnicas de otimização de código no desempenho de uma aplicação em C. O objetivo foi analisar e comparar os efeitos das otimizações de laço (*Loop Interchange*, *Loop Unrolling* e *Loop Tiling*) e de diferentes métodos de alocação de memória (estática e dinâmica).

Para avaliar a performance, foram coletadas métricas de tempo de resposta e contadores de eventos de hardware, como acessos à cache L1 e erros de previsão de desvio (*branch misses*), utilizando o perfilador *perf*. Os resultados se baseiam na média de 10 execuções de cada experimento, dentro de um intervalo de confiabilidade de 95%.

1) Comparação do tempo de resposta dos 8 experimentos



Dados coletados (usados para plotar o gráfico acima):
-Convencional Estatico 8.376 +- 0.224 seconds time elapsed (+- 2.68%)
-Convencional Dinamico 5.960 +- 0.224 seconds time elapsed (+- 3.76%)
-Loop Interchange Estatico 3.6852 +- 0.0179 seconds time elapsed (+- 0.49%)
-Loop Interchange Dinamica 4.3742 +- 0.0133 seconds time elapsed (+- 0.30%)
-Loop Unrolling Estatica 7.784 +- 0.258 seconds time elapsed (+- 3.32%)
-Loop Unrolling Dinamica 6.176 +- 0.221 seconds time elapsed (+- 3.58%)
-Loop Tiling Estatico 4.1110 +- 0.0212 seconds time elapsed (+- 0.52%)
-Loop Tiling Dinamico 5.092 +- 0.101 seconds time elapsed (+- 1.99%)

A análise dos tempos de execução mostra que as otimizações de código melhoraram significativamente a performance. A técnica mais eficaz foi o **Loop Interchange com alocação estática**, que registrou tempo de **3,69 segundos**. Em comparação, o código **convencional estático** foi o mais lento, com **8,38 segundos**.

As otimizações **Loop Interchange** e **Loop Tiling** foram as que trouxeram os maiores benefícios, enquanto o **Loop Unrolling** teve um resultado menos interessante, estando mais próximo do obtido com os códigos convencionais.

Um ponto interessante foi o efeito da alocação de memória. Para o código não otimizado, a alocação dinâmica foi mais rápida. No entanto, para as otimizações mais eficientes (**Interchange** e **Tiling**), a alocação estática se mostrou superior, sendo possível analisar que o uso de um modelo de memória contínua funciona melhor com algoritmos otimizados para cache.

2) Análise dos fatores nas variáveis de resposta de cache e branch considerando os experimentos [1 e 2] e [7 e 8]

A segunda parte da atividade foi realizada utilizando outro computador, comparando os métodos **Convencional Estático** e **Dinâmico** e **Loop Tiling Estático** e **Dinâmico**. As análises são feitas com base nos dados referentes à "cpu_core" presente nos resultados dos perfs.

Resultado do perf sem uso de otimização com alocação estática:

```
  1.344.981.352      cpu_atom/L1-dcache-loads/                                             ( +-  2,48% )  (0,11%)
  2.164.607.481      cpu_core/L1-dcache-loads/                                             ( +-  0,13% )  (99,89%)
<not supported>      cpu_atom/L1-dcache-load-misses/
  1.061.055.978      cpu_core/L1-dcache-load-misses/  #   49,02% of all L1-dcache accesses   ( +-  0,14% )  (99,89%)
    997.344.166      cpu_atom/branch-instructions/                                         ( +-  9,83% )  (0,11%)
  1.084.750.559      cpu_core/branch-instructions/                                         ( +-  0,14% )  (99,89%)
      6.604.010      cpu_atom/branch-misses/          #    0,66% of all branches           ( +- 55,62% )  (0,11%)
      1.070.146      cpu_core/branch-misses/          #    0,10% of all branches           ( +-  0,26% )  (99,89%)
         2,8085 +- 0,0195 seconds time elapsed  ( +-  0,70% )
```

Resultado do perf sem uso de otimização com alocação dinâmica:

```
    816.161.220      cpu_atom/L1-dcache-loads/                                             ( +- 14,39% )  (1,44%)
  3.241.163.760      cpu_core/L1-dcache-loads/                                             ( +-  0,12% )  (98,56%)
<not supported>      cpu_atom/L1-dcache-load-misses/
  1.088.724.137      cpu_core/L1-dcache-load-misses/  #   33,59% of all L1-dcache accesses   ( +-  0,70% )  (98,56%)
    429.121.206      cpu_atom/branch-instructions/                                         ( +-  8,19% )  (1,44%)
  1.083.877.293      cpu_core/branch-instructions/                                         ( +-  0,09% )  (98,56%)
      5.834.931      cpu_atom/branch-misses/          #    1,36% of all branches           ( +- 34,56% )  (1,44%)
      1.068.684      cpu_core/branch-misses/          #    0,10% of all branches           ( +-  0,12% )  (98,56%)
         0,9802 +- 0,0122 seconds time elapsed  ( +-  1,25% )
```

Resultado do perf com loop tiling com alocação estática:

```
    319.666.517      cpu_atom/L1-dcache-loads/                                   ( +- 28,20% )  (0,08%)
  2.235.619.666      cpu_core/L1-dcache-loads/                                   ( +-  0,27% )  (99,92%)
<not supported>      cpu_atom/L1-dcache-load-misses/
     65.335.203      cpu_core/L1-dcache-load-misses/  #    2,92% of all L1-dcache accesses   ( +-  0,31% )  (99,92%)
    212.859.347      cpu_atom/branch-instructions/                               ( +- 25,29% )  (0,08%)
  1.228.943.434      cpu_core/branch-instructions/                               ( +-  0,25% )  (99,92%)
      3.301.566      cpu_atom/branch-misses/          #    1,55% of all branches           ( +- 32,59% )  (0,08%)
      4.501.251      cpu_core/branch-misses/          #    0,37% of all branches           ( +-  0,29% )  (99,92%)
        0,48595 +- 0,00160 seconds time elapsed  ( +-  0,33% )
```

Resultado do perf com loop tiling com alocação dinâmica:

```
    443.393.005      cpu_atom/L1-dcache-loads/                                   ( +- 32,63% )  (0,09%)
  3.463.881.799      cpu_core/L1-dcache-loads/                                   ( +-  0,24% )  (99,91%)
<not supported>      cpu_atom/L1-dcache-load-misses/
     44.516.055      cpu_core/L1-dcache-load-misses/  #    1,29% of all L1-dcache accesses   ( +-  3,47% )  (99,91%)
    227.422.408      cpu_atom/branch-instructions/                               ( +- 24,35% )  (0,09%)
  1.229.317.258      cpu_core/branch-instructions/                               ( +-  0,22% )  (99,91%)
      2.546.145      cpu_atom/branch-misses/          #    1,12% of all branches           ( +- 29,52% )  (0,09%)
      4.919.152      cpu_core/branch-misses/          #    0,40% of all branches           ( +-  2,23% )  (99,91%)
        0,51991 +- 0,00326 seconds time elapsed  ( +-  0,63% )
```

Obs.: Alguns dos gráficos possuem um erro ortográfico escrito "tilt" quando se refere ao método loop tiling.

Gráfico do efeito dos fatores sobre a quantidade de branch instructions

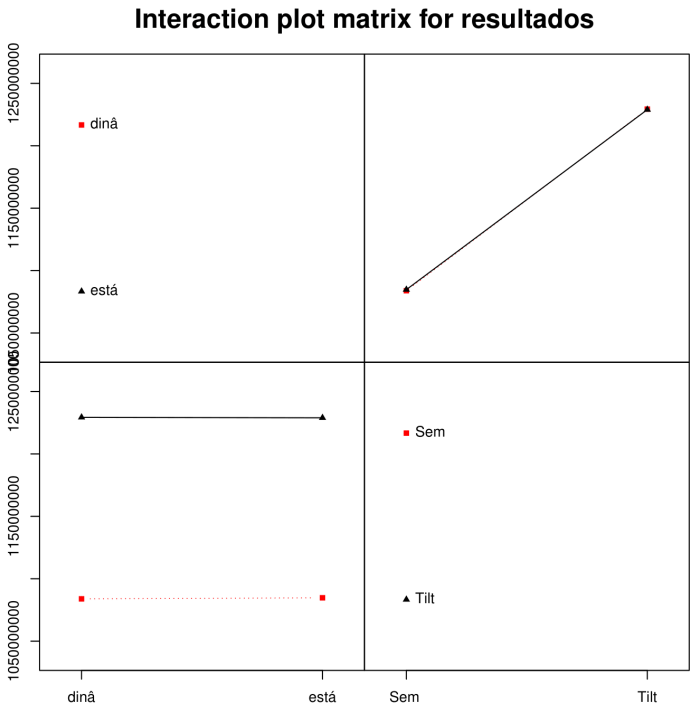


Gráfico do efeito dos fatores sobre a quantidade de branch misses

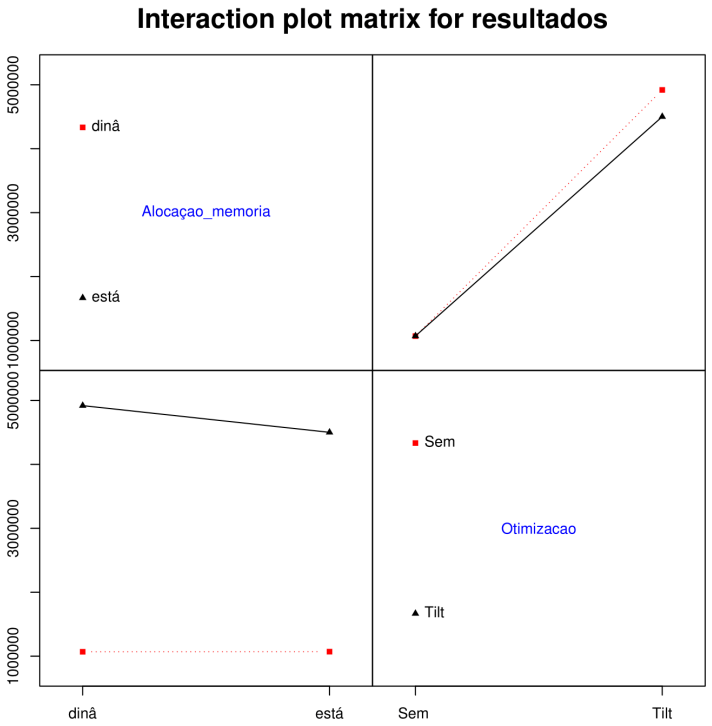


Gráfico do efeito dos fatores sobre a quantidade de cache loads

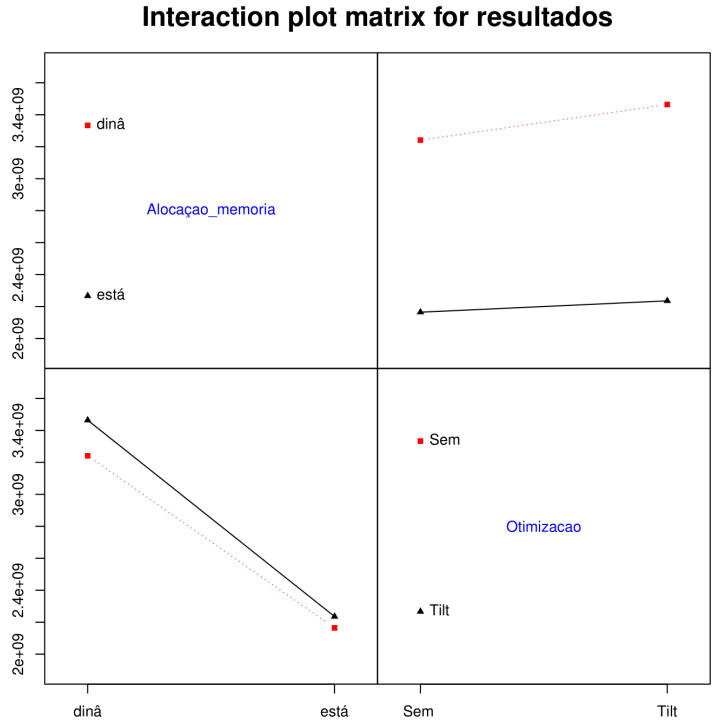
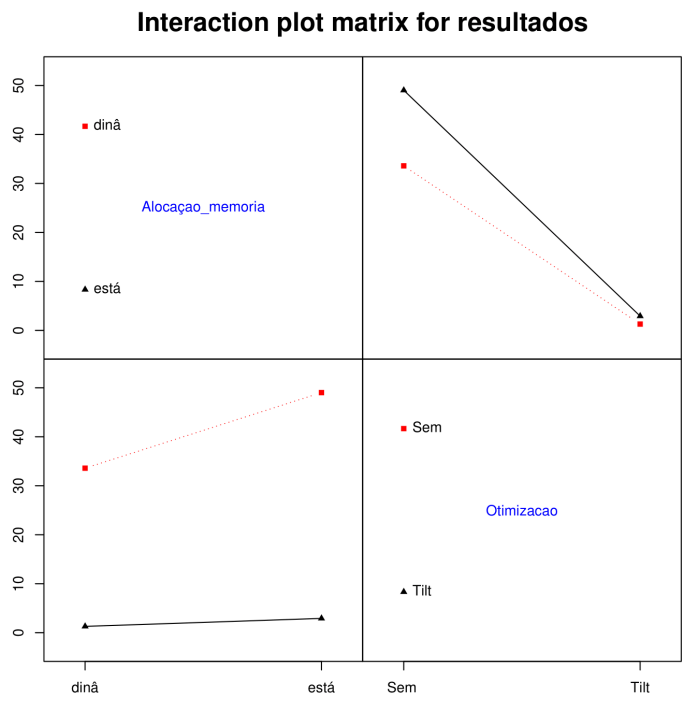


Gráfico do efeito dos fatores sobre a quantidade de cache load misses (em %)



1. Análise de Cache
A otimização **Loop Tiling** teve resultados expressivamente melhores na eficiência do uso da cache L1.
- No código não otimizado, as taxas de *cache miss* eram muito altas: **49,02%** (estático) e **33,59%** (dinâmico). Com a aplicação do Loop Tiling, esses valores despencaram para apenas **2,92%** (estático) e **1,29%** (dinâmico). Essa redução drástica mostra que a técnica melhorou a localidade dos dados, mantendo as informações necessárias na cache e evitando buscas lentas à memória RAM. Essa é a principal razão para o grande ganho de velocidade observado.
- A alocação **dinâmica** consistentemente levou a uma taxa de cache miss ligeiramente menor em ambos os cenários (33,59% vs 49,02% e 1,29% vs 2,92%). O cenário com a melhor performance de cache foi o **Loop Tiling com alocação dinâmica (1,29%)**.

2. Análise de Branch (Previsão de Desvio)
- De forma contraintuitiva, a técnica **Loop Tiling piorou o desempenho da previsão de desvios**. A taxa de branch miss subiu de **0,10%** no código não otimizado para cerca de **0,37% a 0,40%** no código com Tiling. Isso provavelmente ocorre porque a estrutura de laços do Tiling é mais complexa, tornando o padrão de desvios mais difícil para o processador prever corretamente.
- **Efeito da Alocação:** A alocação de memória (estática ou dinâmica) não teve um impacto significativo na performance da previsão de desvios.

Conclusão da Análise
A otimização **Loop Tiling é mais eficaz porque o seu ganho na eficiência de cache supera em muito a pequena perda de performance na previsão de desvios**. Para algoritmos que processam grandes volumes de dados como este, o acesso à memória é o principal gargalo. Portanto, a redução das falhas de cache (de ~40% para ~2%) é o fator dominante que leva à aceleração do programa, mesmo com um leve aumento nos erros de previsão de desvios.