SuperComp - Atividade 5

Giovana Cassoni Andrade.

Utilizando o mesmo código convolução em C++, é feita uma modificação no valor do número de iterações da convolução, aumentando de 15 para 30, 50 e 100, aumentando a complexidade computacional, e compilando com diferentes níveis de otimização.

	15 iterations	**30 iterations**	**50 iterations**	**100 iterations**
basicao	0.232993 s	0.464386 s	0.769176 s	1.54367 s
-01	0.0649907 s	0.128306 s	0.212975 s	0.423785 s
-02	0.173254 s	0.349493 s	0.581074 s	1.15764 s
-03 -march=native	0.0794081 s	0.15807 s	0.258427 s	0.524863 s

Tabela 1 - Tempo de execução para os diferentes valores de iterações da convolução e níveis de otimização.

Para uma comparação visual, foi gerado um gráfico do tempo de execução do algoritmo para as diferentes otimizações com os distintos valores de iterações (Gráfico 1).

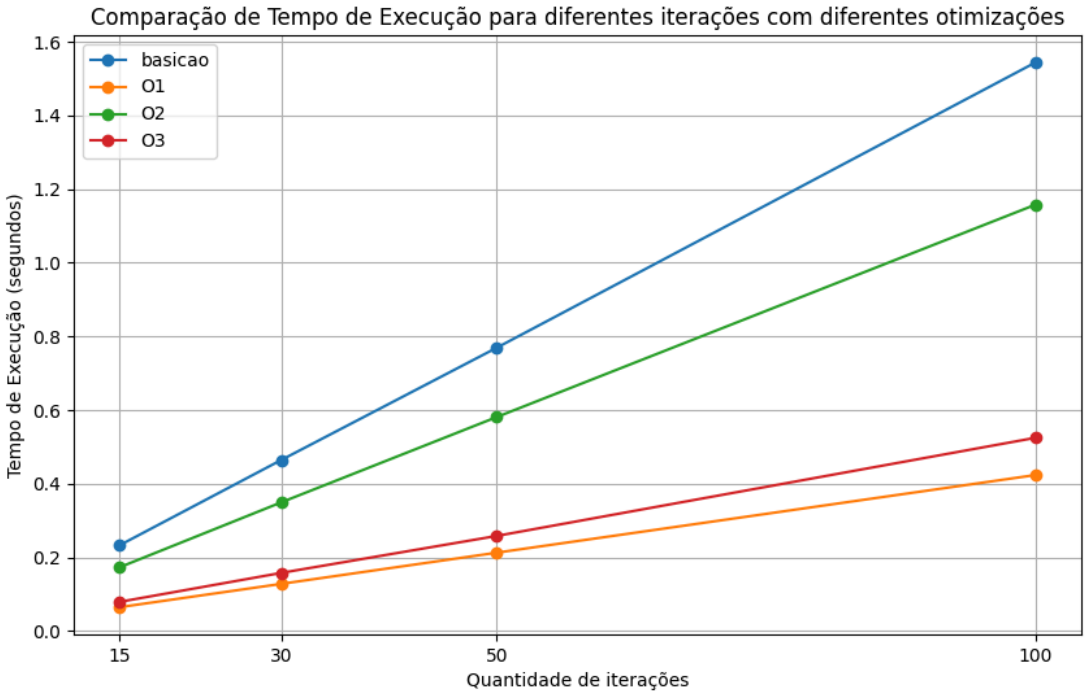

Gráfico 1 - Comparação dos tempos de execução para diferentes iterações.

Ao examinar o gráfico, observa-se que a flag -O1 foi a que mais impactou o desempenho, sendo a mais eficiente com todas as variações de iterações. Por outro lado, a flag -O2, que aplica otimizações mais agressivas, teve um desempenho inferior ao -O1, indicando que as otimizações adicionais não foram tão eficazes para este caso específico. A combinação -O3 -march=native, mesmo melhorando o desempenho em relação ao código básico, ainda assim foi menos eficiente que -O1, mostrando que, para este código, otimizações mais simples foram mais benéficas que as mais avançadas.

Após a compilação com os diferentes níveis de otimização, foi feito um profiling com o Gprof para as diferentes iterações, juntando os resultados do Flat Profile na Figura 1 e os resultados do Call Graph na Figura 2:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
100.00      0.09     0.09 15000000     6.00     6.00  apply_filter(int, int, int (*) [1000], int (*) [5])
```

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
100.00      0.26     0.26 30000000     8.67     8.67  apply_filter(int, int, int (*) [1000], int (*) [5])
```

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 97.67      0.42     0.42 50000000     8.40     8.40  apply_filter(int, int, int (*) [1000], int (*) [5])
  2.33      0.43     0.01                             _init
```

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 90.36      0.75     0.75 100000000     7.50     7.50  apply_filter(int, int, int (*) [1000], int (*) [5])
  6.02      0.80     0.05                             _init
  3.61      0.83     0.03                             main
```

Figura 1 - Comparação dos Flat Profiles de cada uma das iterações realizadas.

```
granularity: each sample hit covers 4 byte(s) for 11.11% of 0.09 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.09                 main [1]
                0.09    0.00 15000000/15000000     apply_filter(int, int, int (*) [1000], int (*) [5]) [2]
-----------------------------------------------
                0.09    0.00 15000000/15000000     main [1]
[2]    100.0    0.09    0.00 15000000         apply_filter(int, int, int (*) [1000], int (*) [5]) [2]
-----------------------------------------------
```

```
granularity: each sample hit covers 4 byte(s) for 3.85% of 0.26 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.26                 main [1]
                0.26    0.00 30000000/30000000     apply_filter(int, int, int (*) [1000], int (*) [5]) [2]
-----------------------------------------------
                0.26    0.00 30000000/30000000     main [1]
[2]    100.0    0.26    0.00 30000000         apply_filter(int, int, int (*) [1000], int (*) [5]) [2]
-----------------------------------------------
```

```
granularity: each sample hit covers 4 byte(s) for 2.33% of 0.43 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     97.7    0.00    0.42                 main [1]
                0.42    0.00 50000000/50000000     apply_filter(int, int, int (*) [1000], int (*) [5]) [2]
-----------------------------------------------
                0.42    0.00 50000000/50000000     main [1]
[2]     97.7    0.42    0.00 50000000         apply_filter(int, int, int (*) [1000], int (*) [5]) [2]
-----------------------------------------------
                                                 <spontaneous>
[3]      2.3    0.01    0.00                 _init [3]
-----------------------------------------------
```

```
granularity: each sample hit covers 4 byte(s) for 1.20% of 0.83 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     94.0    0.03    0.75                 main [1]
                0.75    0.00 100000000/100000000     apply_filter(int, int, int (*) [1000], int (*) [5]) [2]
-----------------------------------------------
                0.75    0.00 100000000/100000000     main [1]
[2]     90.4    0.75    0.00 100000000         apply_filter(int, int, int (*) [1000], int (*) [5]) [2]
-----------------------------------------------
                                                 <spontaneous>
[3]      6.0    0.05    0.00                 _init [3]
-----------------------------------------------
```

Figura 2 - Comparação dos Call Graphs de cada uma das iterações realizadas.

Interpretando o Flat Profile (Figura 1) e o Call Graph (Figura 2) gerados, pode-se ver que a função apply_filter é o ponto de consumo de tempo mais significativo em todos os casos. Como apply_filter é chamada 15, 30, 50 e 100 milhões de vezes, respectivamente, e consome de 100% a 90.36% do tempo de execução, ela é a função crítica, e qualquer otimização dessa função terá um impacto direto e significativo no desempenho geral.

A análise sugere que o foco de otimização deve ser na apply_filter, possivelmente explorando maneiras de reduzir o número de chamadas, otimizar o código interno, ou ambos.

Também é feito um profiling Valgrind com o Callgrind, para as diferentes iterações, na qual pode ser visto na Figura 3.

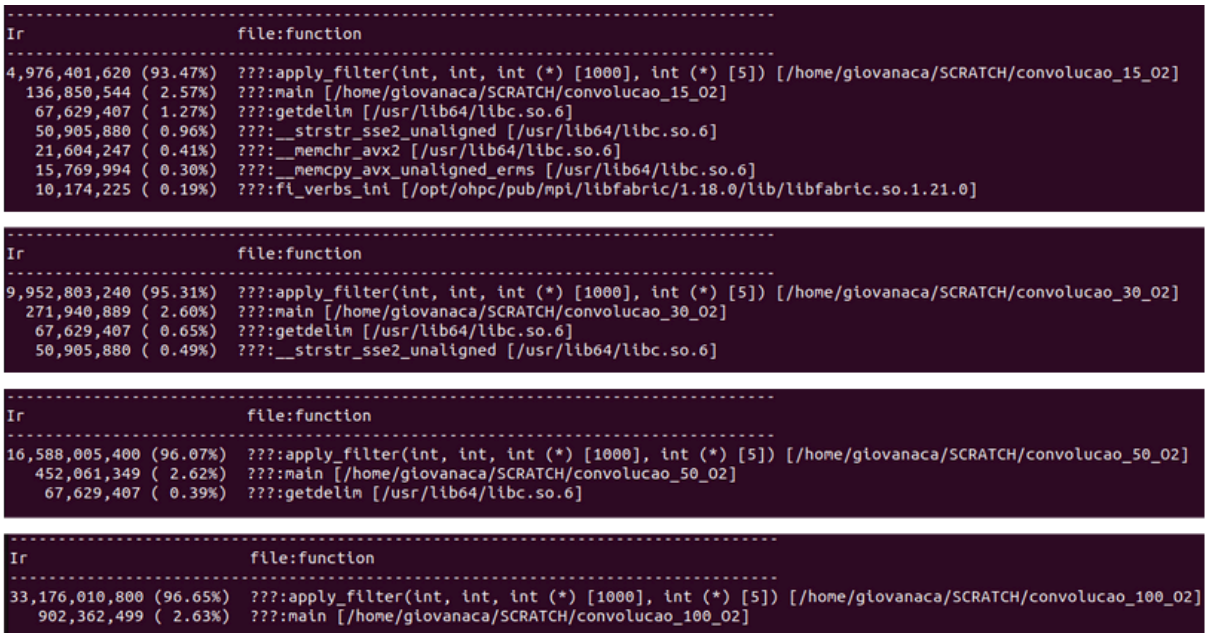

Figura 3 - Comparação dos Valgrind de cada uma das iterações realizadas.

Ao analisar a Figura 3, afirma-se que a função apply_filter é a que possui o maior custo, variando de 93.47% a 96.65% das instruções totais executadas para os códigos. As outras funções main e getdelim consomem apenas uma pequena porção das instruções totais, mas ainda assim são relevantes.

Posto isso, pode-se identificar que uma função candidata para otimização, podendo resultar em melhorias significativas de desempenho, é a apply_filter. Nesta função, estão presentes dois loops aninhados, ambos candidatos para otimização, os quais seria possível otimizar ao trocar a iteração do loop ou simplificar a verificação condicional dos limites da matriz.

Com as análises feitas, ambas as ferramentas Gprof e Callgrind fornecem insights complementares, e a combinação das análises pode orientar a otimização de maneira mais robusta, focando tanto na redução do tempo de CPU quanto na melhoria da eficiência do uso de cache e execução de instruções.

Por fim, algumas otimizações adicionais que poderiam ser feitas são a verificação da paralelização com OpenMP, para ver se realmente está aumentando o desempenho, e evitar operações desnecessárias, como não copiar os dados se a matriz result não é usada entre iterações.