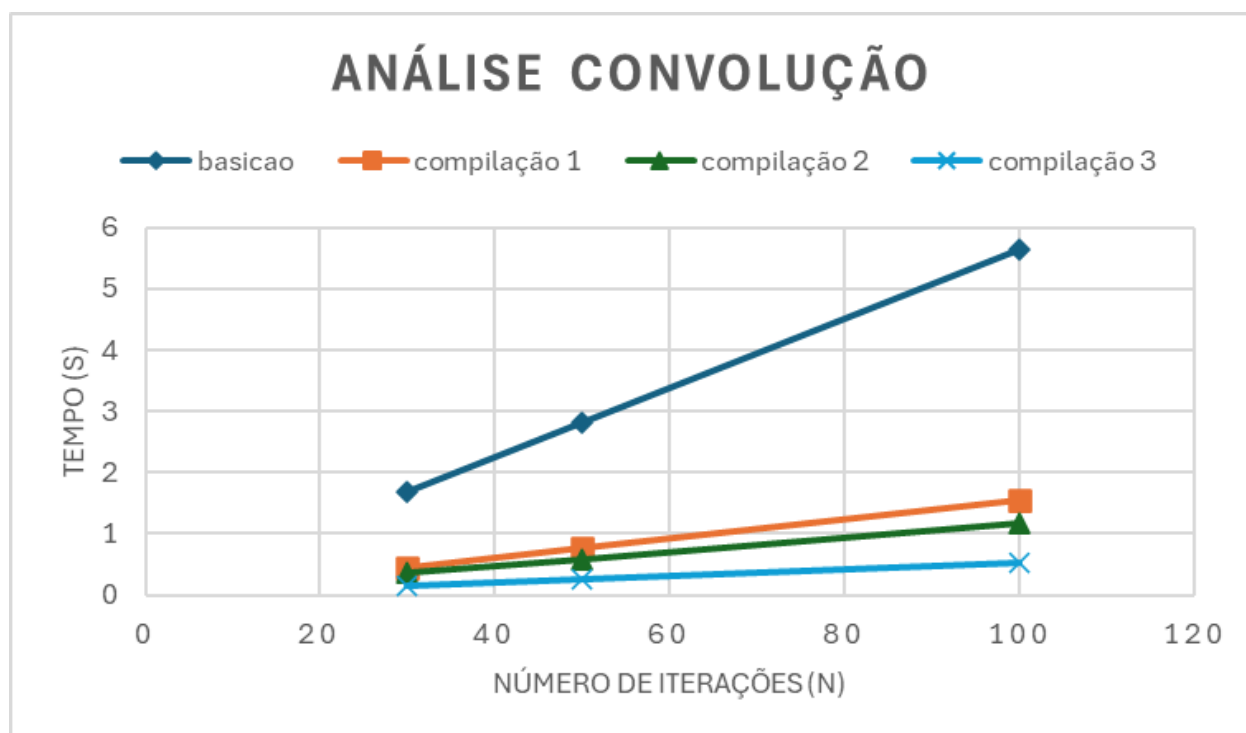


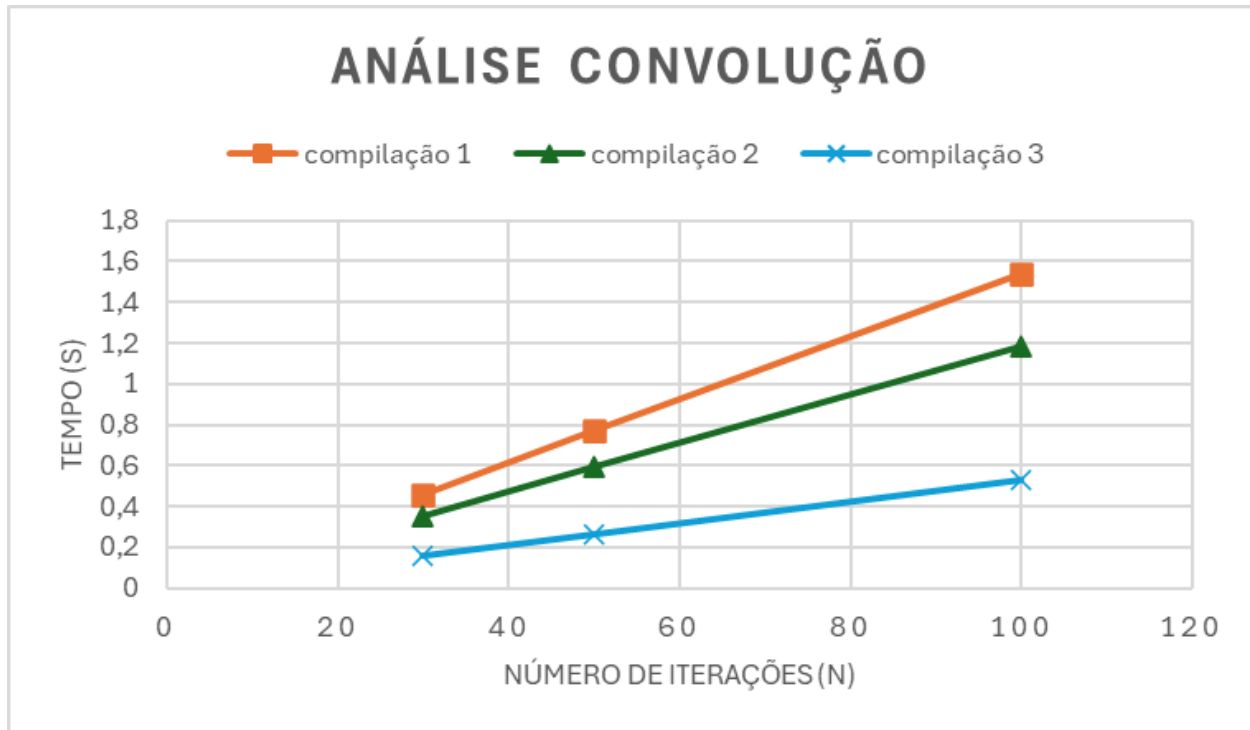
Relatório 5- Bruna Meinberg

Análise dos tempos das diferentes compilações

Utilizando os diferentes tipos de compilação para o código de convolução, encontrei o seguinte tempo:



Visivelmente, o básico tem o desempenho pior, logo, utilizando somente os dados das compilações “diferentes”.



Utilizando a otimização avançada, é perceptível que o tempo é mais curto.

gprof e Callgrind

Compilando utilizando o gprof, o tempo levado para as 100 iterações foi o mesmo tempo da otimização básica. Já o callgrind não conseguiu rodar.

gprof

Analisando a “saída” do gprof, observamos o seguinte:

Each sample counts as 0.01 seconds.

```
% cumulative self      self  total
```

```
time seconds seconds  calls ns/call ns/call name
```

```
91.01   0.81   0.81 1000000000   8.10   8.10 apply_filter(int, int, int () [1000], int () [5])
6.74    0.87   0.06                      main
2.25    0.89   0.02                      _init
```

A maior tempo de código é gasto na chamada da função “apply_filter”, que é chamada muitas vezes.

Callgrind

Rodando os comandos:

```
[brunalm@sms-host aula05]$ valgrind --tool=callgrind ./convolucao_02
==631985== Callgrind, a call-graph generating cache profiler
==631985== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==631985== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==631985== Command: ./convolucao_02
==631985==
==631985== For interactive control, run 'callgrind_control -h'.
A tarefa levou 237.189 segundos para ser executada
==631985==
==631985== Events      : Ir
==631985== Collected : 34327096345
==631985==
==631985== I    refs:      34,327,096,345
```

```
[brunalm@sms-host aula05]$ callgrind_annotate callgrind.out.631985
-----
Profile data file 'callgrind.out.631985' (creator: callgrind-3.22.0)
-----
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 8339813869
Trigger: Program termination
Profiled target: ./convolucao_02 (PID 631985, part 1)
Events recorded: Ir
Events shown: Ir
Event sort order: Ir
Thresholds: 99
Include dirs:
User annotated:
Auto-annotation: on

-----
Ir
-----
34,327,096,345 (100.0%) PROGRAM TOTALS
-----

Ir          file:function
-----
33,176,010,800 (96.65%) ???:apply_filter(int, int, int (*) [1000], int (*) [5]) [/home/brunalm/SCRATCH/supercomputacao/aula05/convolucao_02]
 902,362,499 ( 2.63%) ???:main [/home/brunalm/SCRATCH/supercomputacao/aula05/convolucao_02]
```

Com isso vemos que a função apply_filter toma 96,65% do tempo, dando indicações de onde podemos otimizar.

Sugestões de otimização

De acordo com os resultados, seria interessante agir com otimizações em cima da função `apply_filter`, que com certeza é a mais chamada.

```
int apply_filter(int x, int y, int matrix[N][N], int filter[FILTER_SIZE]) {
    int result = 0;
    int filter_offset = FILTER_SIZE / 2; // Calcula o deslocamento

    // Aplica o filtro 5x5 ao elemento (x, y) da matriz.
    for (int i = -filter_offset; i <= filter_offset; i++) {
        for (int j = -filter_offset; j <= filter_offset; j++) {
            int xi = x + i;
            int yj = y + j;
            // Verifica se o índice está dentro dos limites da matriz.
            if (xi >= 0 && xi < N && yj >= 0 && yj < N) {
                result += matrix[xi][yj] * filter[i + filter_offset + j];
            }
        }
    }
    return result; // Retorna o valor convoluído.
}
```

Provavelmente, devido ao tamanho da matriz (1000x1000 no caso dessa compilação), essa função acaba sendo chamada muitas vezes e, contendo dois loops alinhados, acaba tomando muito tempo.