Universidade Presbiteriana Mackenzie  
Ciência da Computação – 05P11  
Computação Paralela  
20/02/2025

Alan Meniuk Gleizer  
RA 10416804

Atividade – Monitoramento de Processos

**1. Crie dois executáveis para a parte de multiplicação de matrizes: um que percorre em ordem de linha e outro que percorre em ordem de coluna.**

Código fonte nos arquivos:

mult\_mat\_linhas.cpp  
mult\_mat\_colunas.cpp

Neste mesmo repositório.

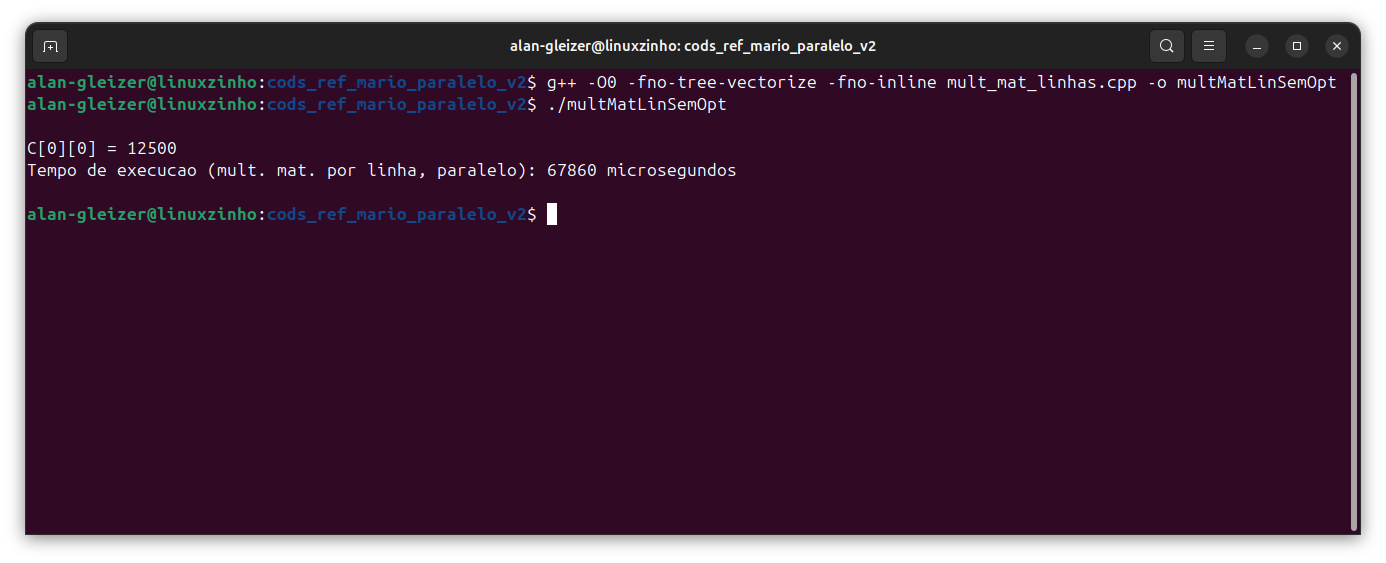
**2. Crie um terceiro executável para utilizar corretamente o cache (hierarquia de memória).**

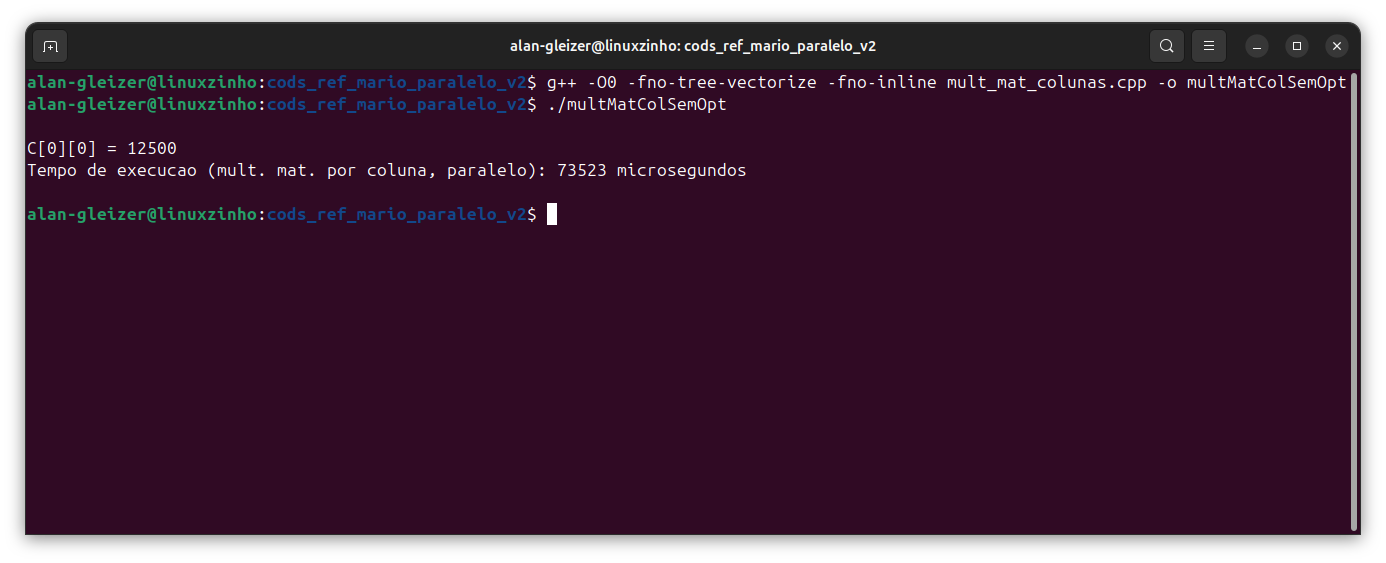
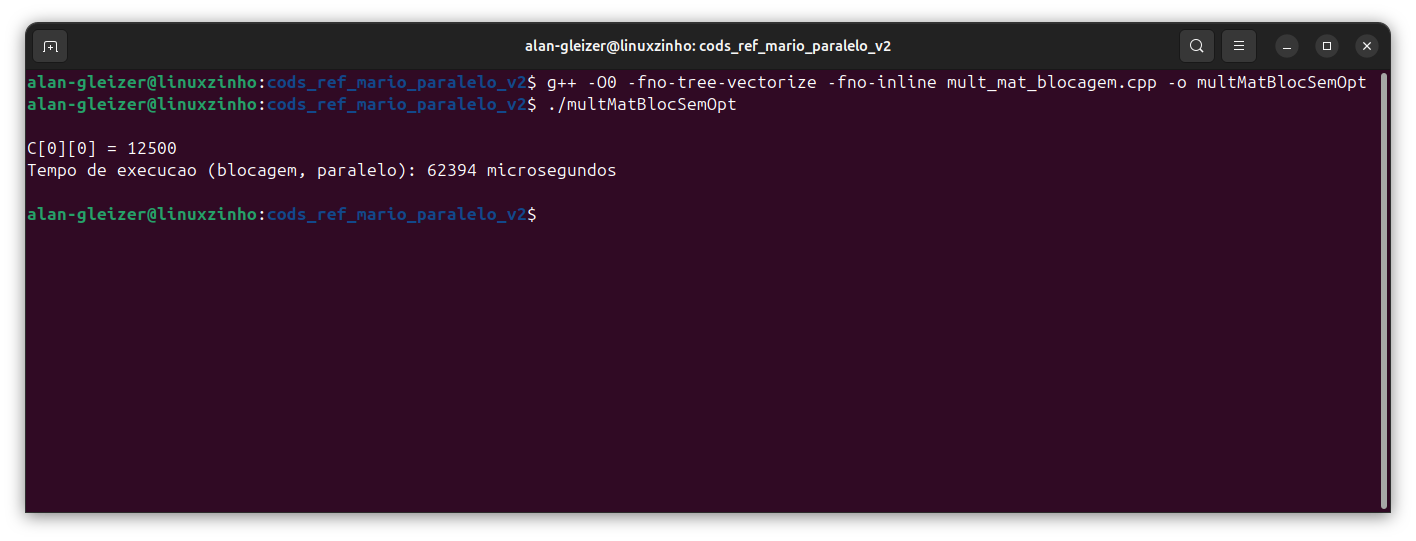
Código fonte no arquivo:

mult\_mat\_blocagem.cpp

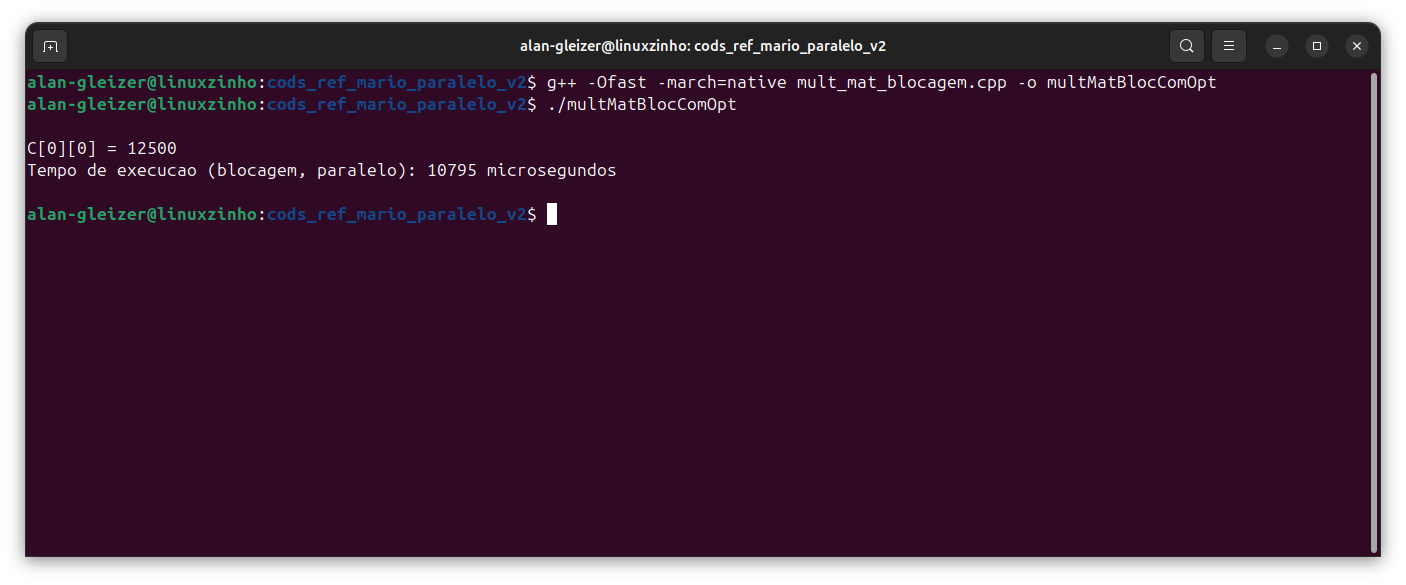
Neste mesmo repositório.

**Para cada opção, você deve medir o tempo de execução:**

**a) desligando todas as otimizações do compilador.**

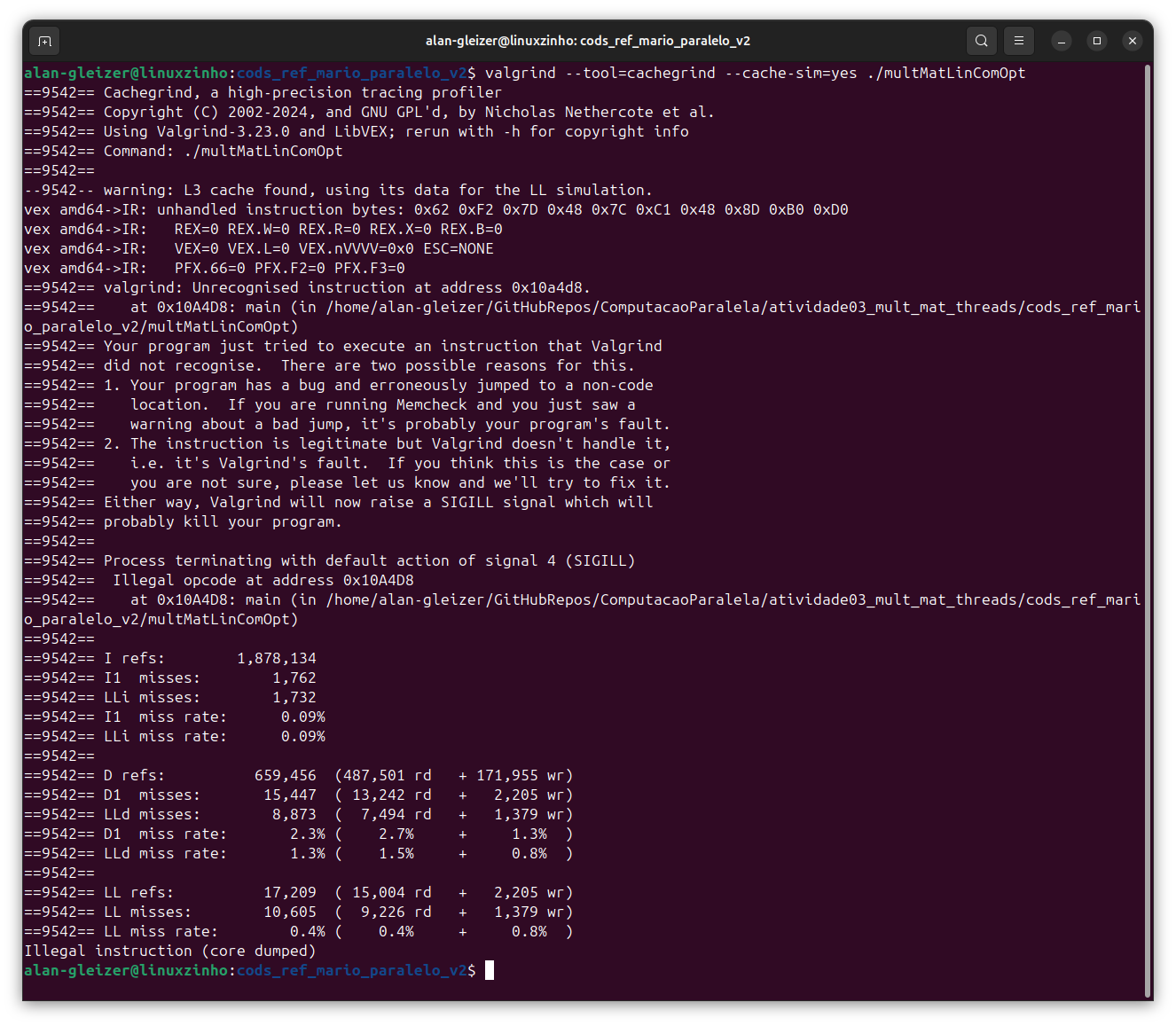


**b) ligando a otimização máxima.**

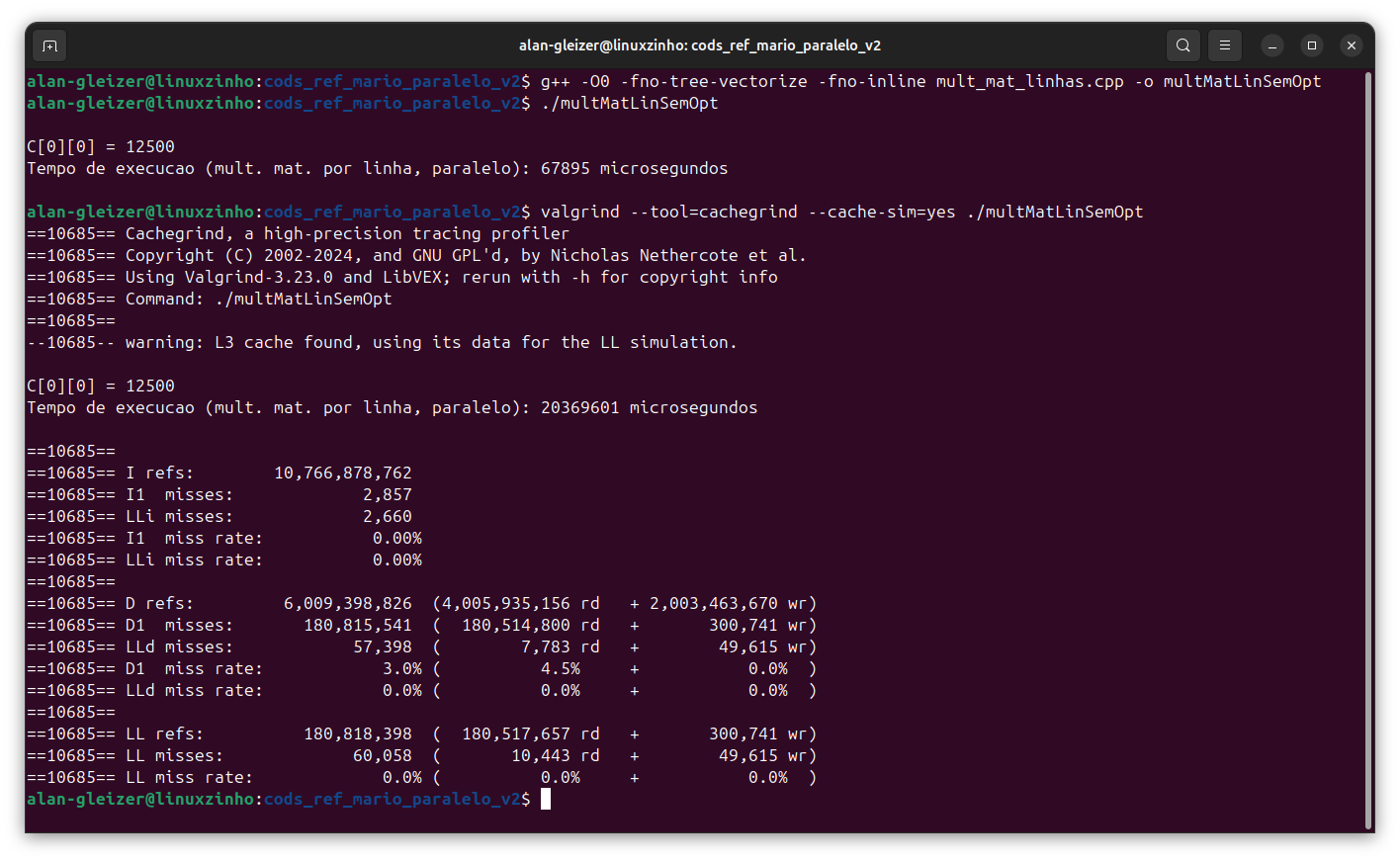
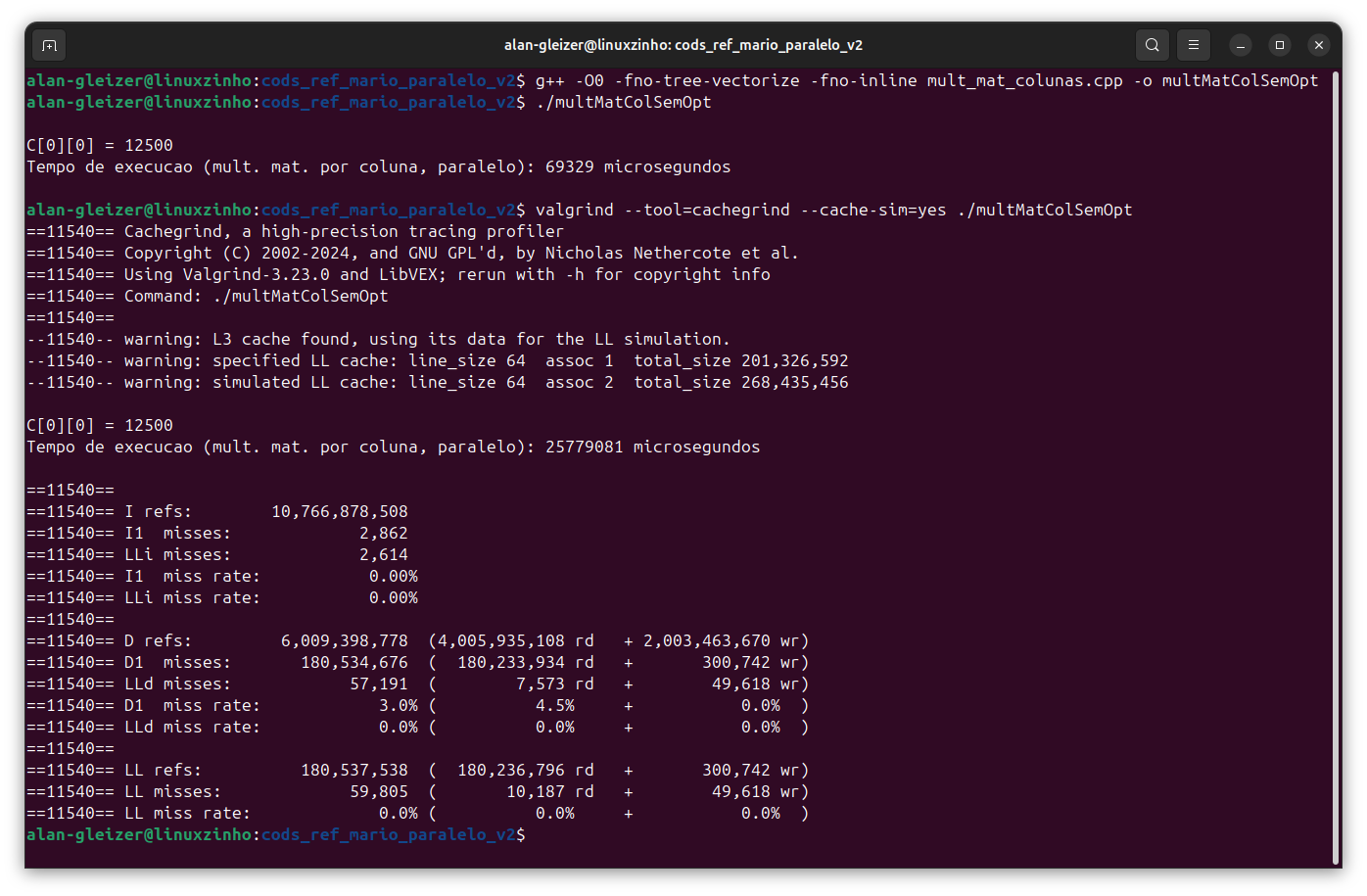


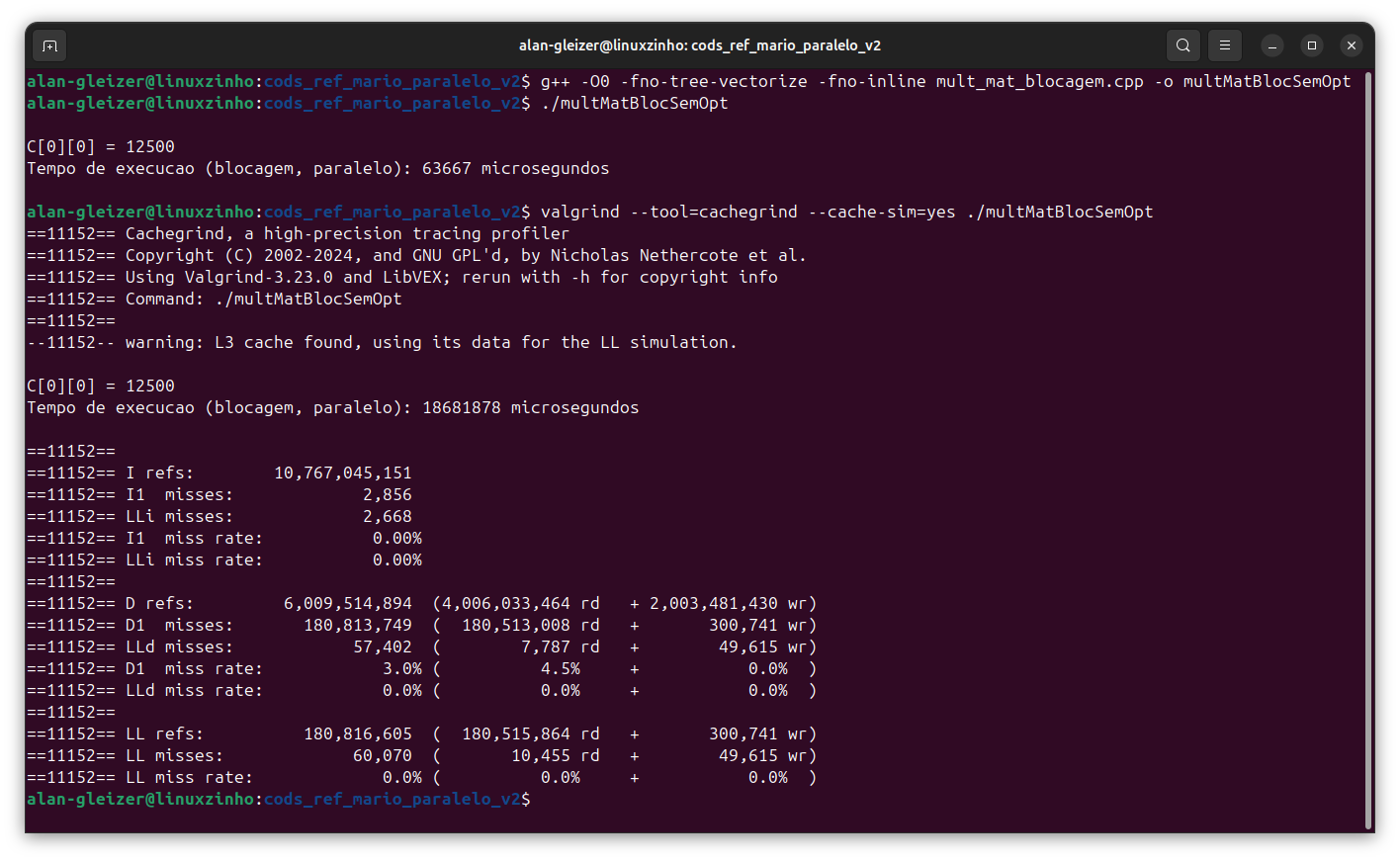
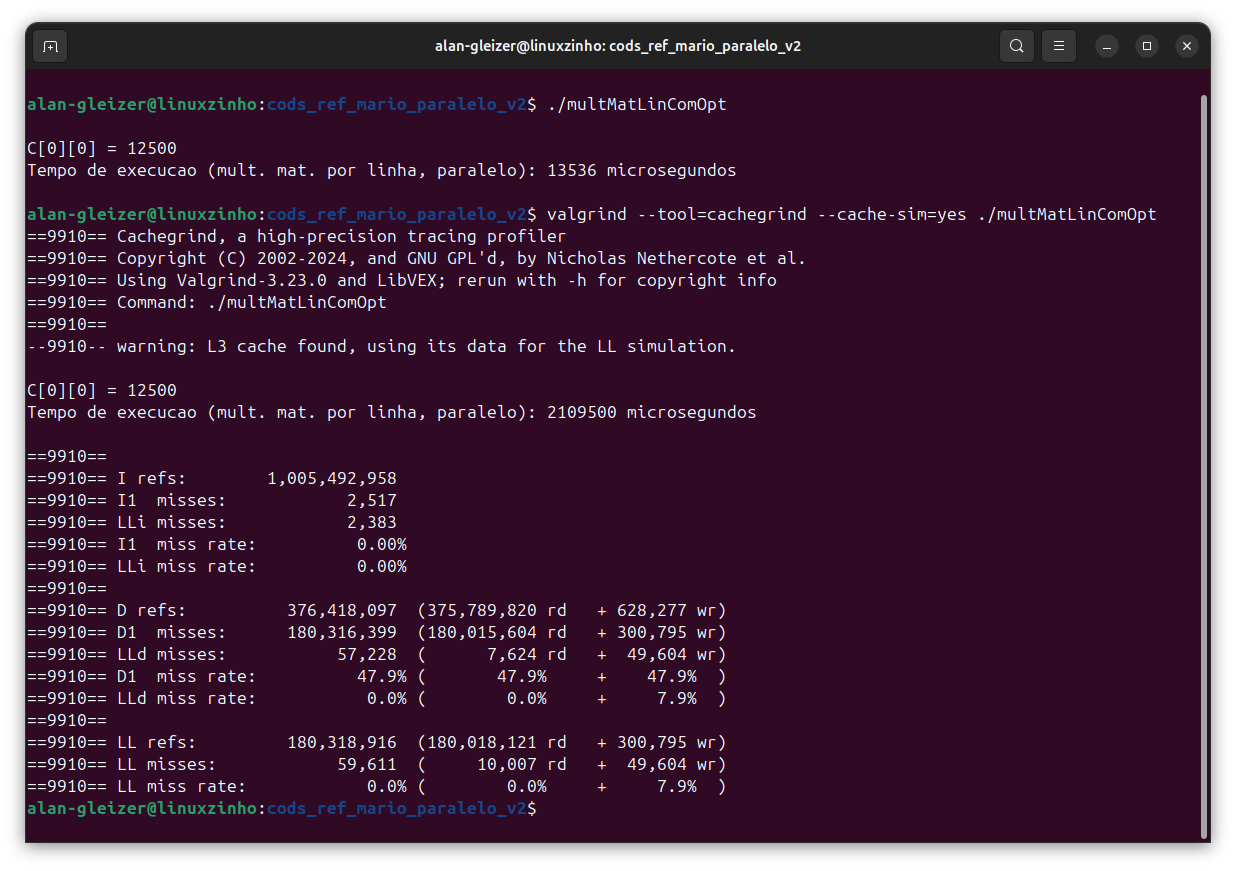
**3. Faça também uma análise do padrão de acesso ao cache de todas as versões utilizando o utilitário valgrind.**

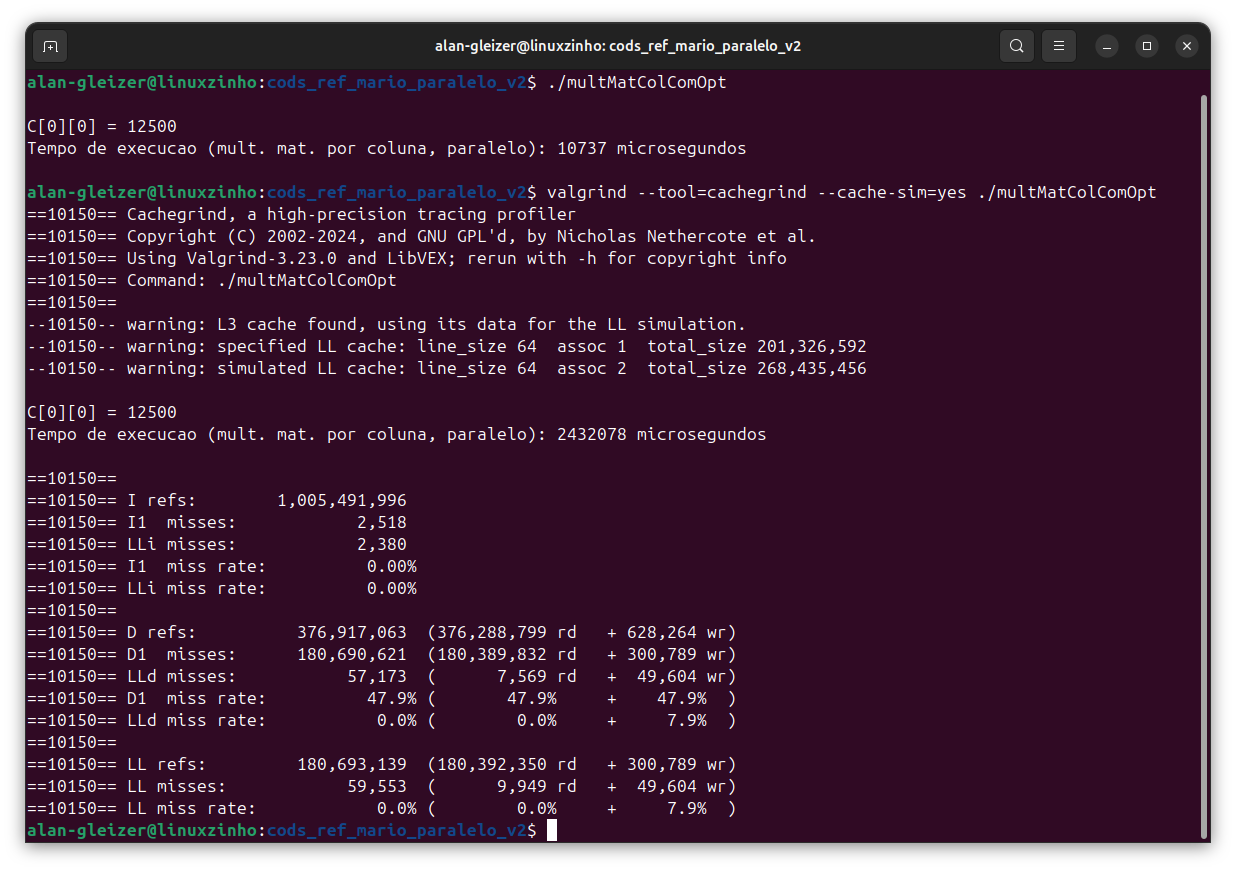
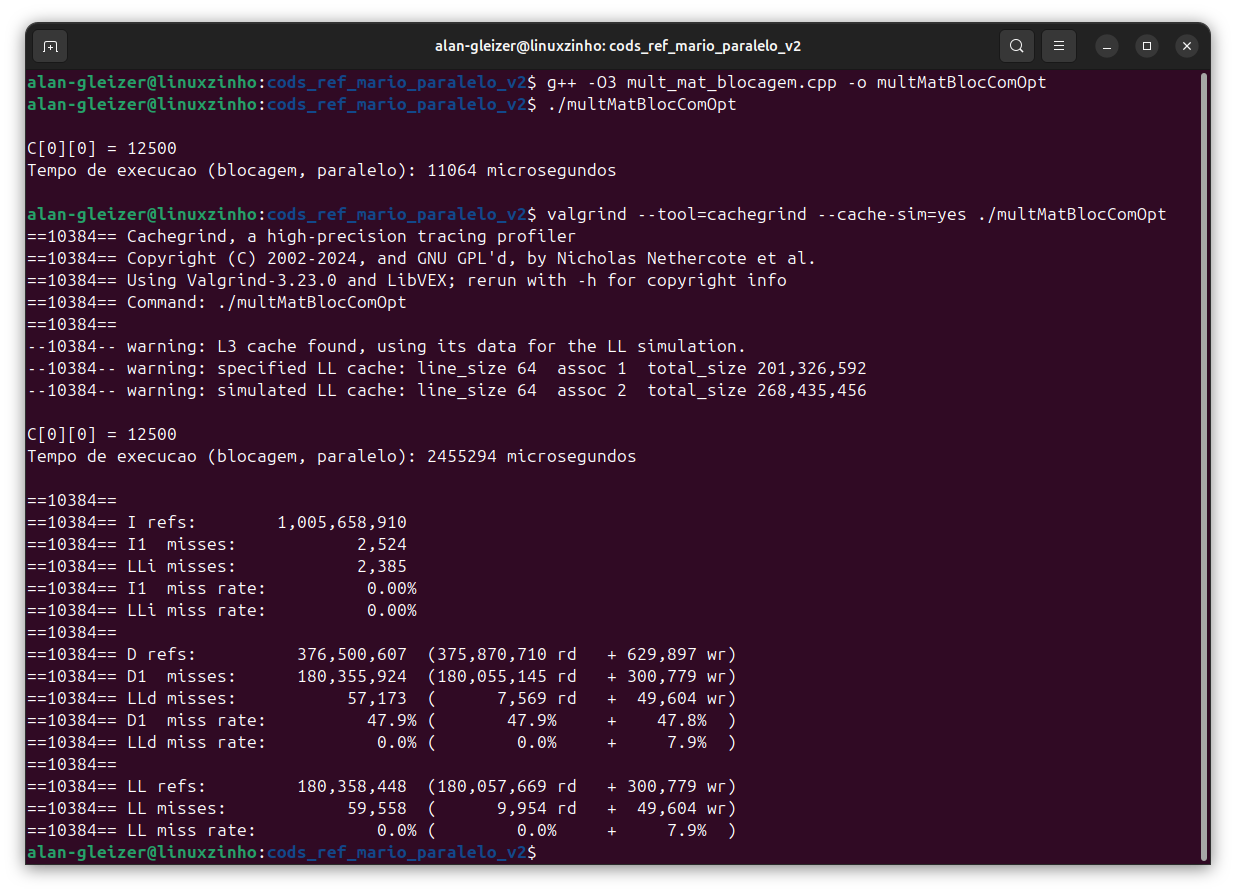
Neste ponto é importante relatar que não foi possível executar o valgrind para os executáveis compilados com as flags -Ofast e -march=native. A primeira se trata de uma flag que ativa -O3, o maior nível de otimização, e ainda ignora conformidade com alguns padrões de representação para ganhar ainda mais velocidade. A segunda flag, por sua vez, ativa compilação com instruções específicas da CPU para ainda mais otimização. Nestes casos, o valgrind detecta uma instrução não-reconhecida no programa e aborta a execução, como no exemplo abaixo:



É provável que se trate de uma incompatibilidade do valgrind com instruções específicas da plataforma AMD Zen 4, na qual os programas foram executados. Dessa forma, foi necessário recompilar as versões otimizadas utilizando a flag -O3 e omitindo - march=native. Os resultados são apresentados abaixo:







**Comentários**

O exercício proposto foi bastante desafiador em múltiplas frentes de aprendizado. EM primeiro lugar, se tratou do segundo contato com a linguagem C++, mas com sintaxe mais avançada do que o visto na disciplina de SO. Ainda que C++ apresente uma série de similaridades com C, as diferenças são suficientes para dificultar a legibilidade do código e o seu entendimento. Em particular é possível citar os operadores << e ::, o uso de objetos e generics com sintaxe distinta tanto de C quanto de Java. Após estudo das estruturas e particularidades de C++, foi possível enfrentar o desafio seguinte: a programação paralelizada em si.

A paralelização dos algoritmos de multiplicação de matrizes foi facilitada pelos exemplos fornecidos no Jupyter Notebook da aula. A estrutura básica das abordagens por linha, por coluna e com blocagem foi mantida, mas em todos os casos foi necessário estruturar uma nova abordagem para implementação dos códigos. Pessoalmente, essa nova abordagem me lembrou o tipo de análise necessária para resolver um problema utilizando recursividade. Foi necessário, por exemplo, entender que seria necessário passar os índices das linhas/colunas que cada thread trabalharia como parâmetros das funções, da mesma forma que utilizamos parâmetros em recursividade para iterar sobre arrays/matrizes. Da mesma forma, foi necessário desenvolver código para definir como seria feita a divisão das linhas/colunas e como armazenar essas informações para as várias interações de loop nos quais executamos as diferentes threads. No caso específico da blocagem, o processo foi ainda mais difícil.

Por fim, houve muita dificuldade no momento de compilação e análise com o Valgrind. De acordo com as pesquisas realizadas, as flags -O0 e -Ofast representam os níveis mínimos e máximos de otimização no g++. Também ficou claro que é recomendado utilizar -march=native para aproveitar particularidades de optimização da arquitetura de cada processador específico. Foi necessário bastante pesquisa e debug para entender, ou desenvolver a hipótese, que algumas dessas flags resultam em instruções que não são suportadas pelo Valgrind.

Compilando os resultados, é possível ver uma série de padrões interessantes:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Programa | | Tempo exec. (μs) | Análise Valgrind  (métricas que apresentaram variabilidade) | | | |
| Num. instruções | D1 miss rate (%) |  |  |
| Ordem coluna | -O0 | 69329 | 10,8 \* 109 | 3,0 |  |  |
| -O3 | 10737 | 1,0 \* 109 | 47,9 |  |  |
| -Ofast  -march=native | 10558 | - | - |  |  |
| Ordem linha | -O0 | 67895 | 10,8 \* 109 | 3,0 |  |  |
| -O3 | 13536 | 1,0 \* 109 | 47,9 |  |  |
| -Ofast  -march=native | 10767 | - | - |  |  |
| Por blocagem | -O0 | 63667 | 10,8 \* 109 | 3,0 |  |  |
| -O3 | 11064 | 1,0 \* 109 | 47,9 |  |  |
| -Ofast  -march=native | 10795 | - | - |  |  |

As otimizações de compilador (-O3, -Ofast) são claramente bastante eficazes, reduzindo o número de operações am aproximadamente 10 vezes e o tempo de execução em quase 6x. Conforme a pesquisa realizada sobre ILP, isso se deve ao fato de que o compilador reorganiza o código aplicando técnicas como, unrolling de loops e pré-buscas de dados. Isso faz com que o programa execute muito mais rápido, pois a CPU realiza menos instruções para cumprir a mesma tarefa. Vale notas que embora a taxa de “cache misses” suba, o tempo total ainda cai, pois a redução no número de instruções e o uso eficiente do pipeline compensam essas falhas.

Nos cenários de “ordem de linha”, “ordem de coluna” ou “blocagem”, sem otimização (-O0), blocagem tende a se sair um pouco melhor graças à localidade de dados. Mas ao usar -O3 ou -Ofast, todas as versões chegam perto de um mesmo patamar de desempenho.