UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

NICOLAS VON DOLINGER MOREIRA ROCHA

ESTRUTURA DE DADOS

RELATÓRIO PRÁTICA 4

BELO HORIZONTE

2023

Introdução

Este relatório descreve a caracterização da localidade de referência no código fornecido para análise. O objetivo é avaliar qualitativamente o comportamento do código em termos de acesso à memória e identificar as estruturas de dados e segmentos de código críticos que afetam o desempenho. O código fornecido é uma aplicação de exemplo que realiza operações matriciais, incluindo multiplicação, soma e transposição de matrizes. O código foi escolhido para análise devido à sua complexidade de acesso à memória, que oferece uma oportunidade valiosa para caracterizar a localidade de referência.

Análise Qualitativa

1 - Acessos de Memória Esperados:
O código realiza intensivos acessos à memória devido às operações em matrizes, especialmente durante a multiplicação de matrizes de grande porte. Além das operações de matrizes, o código também pode acessar a memória para outras operações, como alocação de recursos e manipulação de estruturas de dados complexas.

2 - Localidade de Referência:
O código provavelmente exibirá uma localidade de referência espacial, uma vez que as operações em matrizes geralmente envolvem acesso a elementos próximos na memória. A localidade de referência temporal pode ser observada em iterações sobre matrizes, onde os mesmos elementos de matriz são acessados repetidamente.

3 - Estruturas de Dados Críticas:
As estruturas de dados cruciais que têm o potencial de ter um impacto substancial no desempenho incluem as matrizes de entrada e saída, que representam os principais elementos de acesso à memória. No contexto deste código, a estrutura de dados crítica é denominada 'mat_tipo'. Essa estrutura contém um arranjo bidimensional de números em ponto flutuante e é considerada crítica, visto que

desempenha um papel central em todas as operações de acesso à memória realizadas ao longo da execução do código.

4 - Segmentos de Código Críticos:
Os loops aninhados que lidam com a inicialização aleatória da matriz e o acesso aos seus elementos desempenham um papel crucial na execução do programa. Eles são responsáveis por percorrer todos os elementos da matriz, o que pode resultar em uma demanda intensa de memória e tempo de execução. Essa parte do código é especialmente sensível ao tamanho da matriz, pois o número de iterações aumenta quadraticamente com o tamanho da matriz.
A função 'transpoeMatriz' é outro ponto crítico no código devido ao seu elevado número de operações de acesso à matriz e à complexidade de efetuar trocas de elementos para realizar a transposição. A transposição de uma matriz envolve a troca de elementos entre linhas e colunas, e isso requer um número significativo de operações. A função 'multiplicaMatrizes' apresenta uma estrutura complexa com três loops aninhados e inúmeras operações de acesso à matriz. A multiplicação de matrizes é uma operação computacionalmente intensiva, e o desempenho dessa função pode ser crítico, especialmente para matrizes grandes.

Plano de Caracterização de Localidade de Referência

O objetivo deste plano é avaliar a localidade de referência no código fornecido para identificar oportunidades de otimização em relação ao acesso à memória e ao desempenho geral do programa.

Etapa 1: Compreensão do Código e Identificação de Pontos Críticos

Nesta etapa, foi feita uma revisão do código-fonte para entender sua estrutura, operações principais e estruturas de dados relevantes. Foram identificados os pontos críticos do código que podem afetar o acesso à memória. É importante ter uma compreensão clara do programa e das áreas potencialmente problemáticas antes de iniciar a caracterização.

Etapa 2: Seleção de Cenários de Execução

Nesta etapa, foram escolhidos os cenários de execução representativos que envolvam operações críticas de acesso à memória. Isso pode incluir operações de multiplicação de matrizes, soma de matrizes ou outras operações significativas. A escolha de cenários de execução representa o comportamento real do programa e direciona a caracterização para áreas críticas de acesso à memória.

Etapa 3: Execução do Cachegrind para Caracterização

Usaremos a ferramenta Cachegrind (parte do Valgrind) para caracterizar a localidade de referência durante a execução do programa nos cenários selecionados. Executaremos o programa com os seguintes comandos:

valgrind --tool=cachegrind ./MeuPrograma <cenário_de_execução>

O Cachegrind fornece informações detalhadas sobre o comportamento do acesso à memória, incluindo métricas de cache e taxa de acertos, ajudando a identificar falhas de cache e áreas de otimização.

Etapa 4: Análise dos Resultados do Cachegrind

Após a execução do programa com o Cachegrind, analisaremos os resultados gerados pelo Cachegrind usando a ferramenta **cg_annotate**. Isso nos permitirá entender o comportamento de acesso à memória e identificar áreas críticas que requerem otimização. A análise dos resultados nos fornece insights sobre onde ocorrem os principais acessos à memória e onde estão os gargalos de desempenho.

O mesmo será feito com o Callgrind.

Parâmetros

Nesta seção do relatório, descreveremos os parâmetros selecionados para a caracterização do código, conforme definido no Makefile fornecido. As seguintes execuções foram realizadas:

1. **Multiplicação de Matrizes (500x500):**

   Comando: **$(EXE) -m -p /tmp/mult500.out -x 500 -y 500**

Descrição: Esta execução representa uma multiplicação de matrizes de dimensão 500x500. É uma operação computacionalmente intensiva que envolve o acesso a muitos elementos da matriz.

2. **Soma de Matrizes (500x500):**

   Comando: **$(EXE) -s -p /tmp/soma500.out -x 500 -y 500**

   Descrição: Esta execução representa uma operação de soma de matrizes de dimensão 500x500. Também envolve acesso intensivo à memória para adição de elementos de matriz.

3. **Transposição de Matrizes (500x500):**

   Comando: **$(EXE) -t -p /tmp/transp500.out -x 500 -y 500**

   Descrição: Nesta execução, a transposição de matrizes de dimensão 500x500 é realizada. A transposição envolve reorganização de dados na memória e é crítica para a localidade de referência.

4. **Multiplicação de Matrizes (100x100) com Registro de Acesso:**

   Comando: **$(EXE) -m -p /tmp/mult100log.out -l -x 100 -y 100**

   Descrição: Esta execução é semelhante à primeira, mas também inclui registro de acesso (-l). Isso permitirá uma análise detalhada dos padrões de acesso à memória durante a multiplicação de matrizes menores de 100x100.

Justificativa para a Seleção dos Parâmetros:

Os parâmetros selecionados incluem operações críticas de acesso à memória, como multiplicação, soma e transposição de matrizes. Essas operações envolvem leitura e escrita intensivas na memória, tornando-as relevantes para a caracterização de localidade de referência. A escolha de matrizes de dimensão 500x500 e 100x100 permite avaliar o comportamento do programa em diferentes escalas, abrangendo operações de grande e pequeno porte.

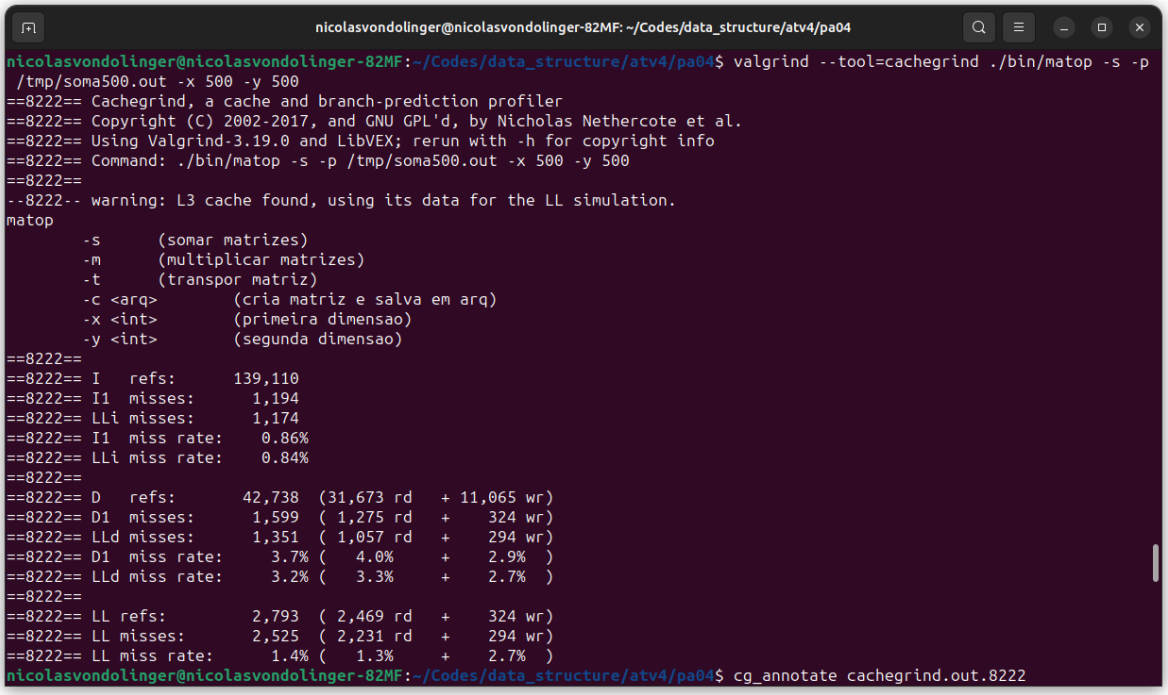
A execução com registro de acesso (opção -l) na multiplicação de matrizes de 100x100 permitirá uma análise detalhada dos padrões de acesso à memória

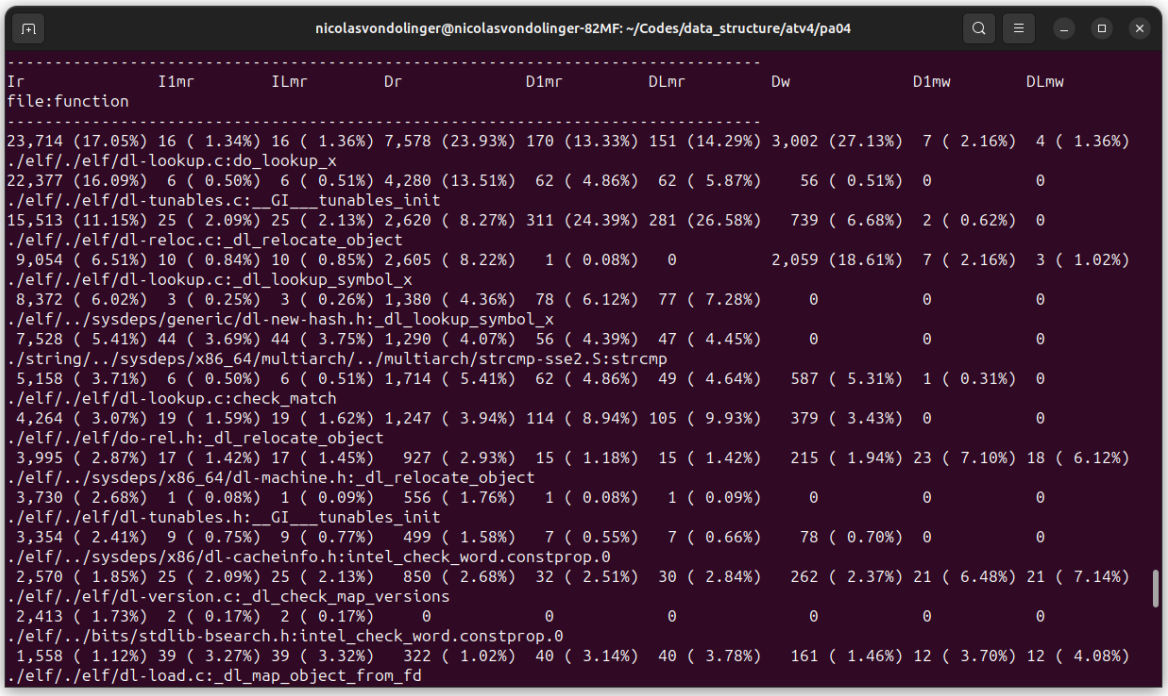
durante essa operação específica. As operações selecionadas refletem situações comuns em programas que lidam com manipulação de matrizes, e os resultados obtidos ajudarão a identificar áreas críticas para otimização em termos de localidade de referência.

Nesse sentido, esses parâmetros foram escolhidos com base na complexidade de acesso à memória que eles representam e na importância de entender como o programa se comporta durante essas operações críticas.

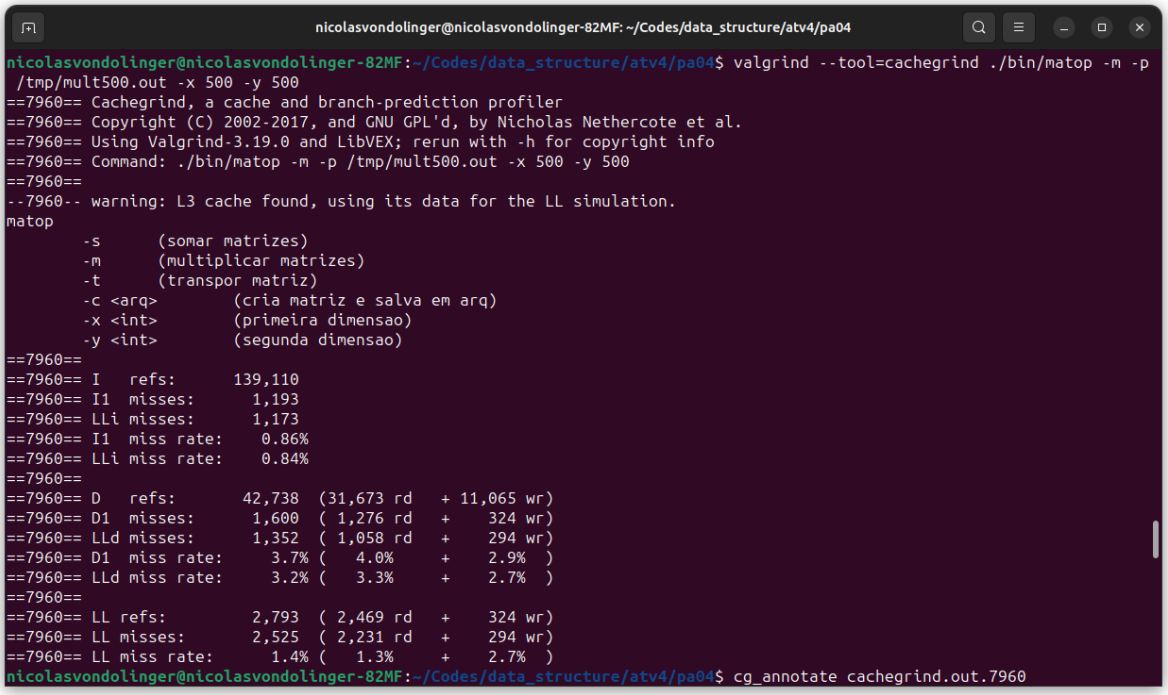
Cachegrind

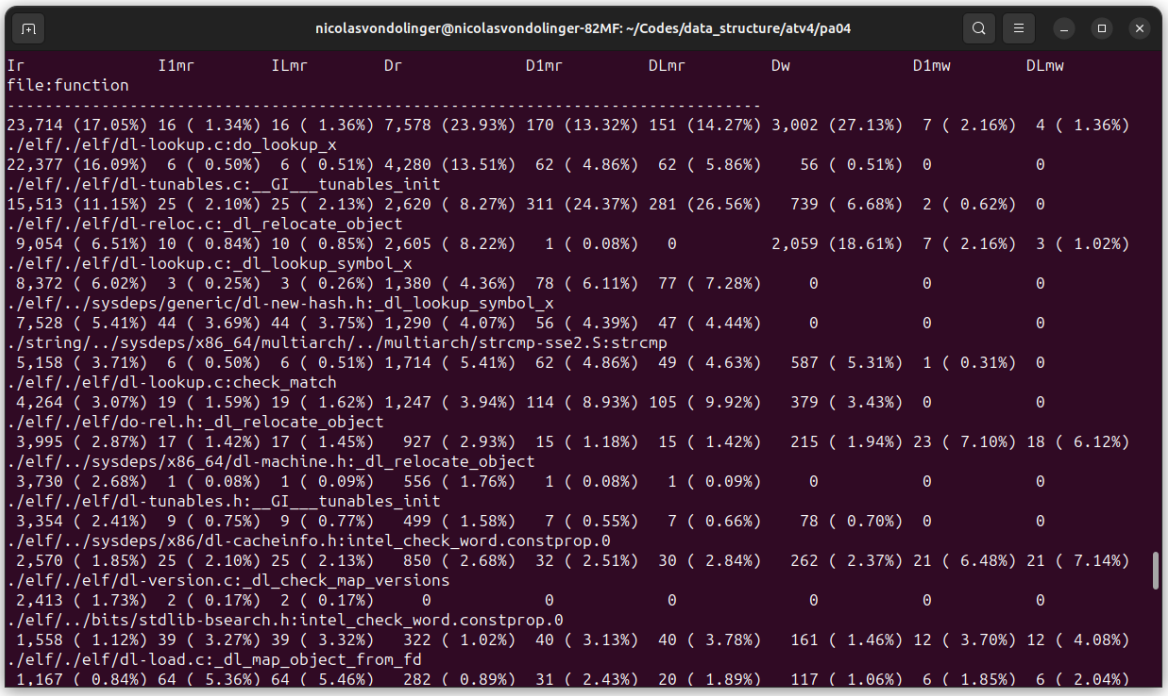
Soma de Matrizes



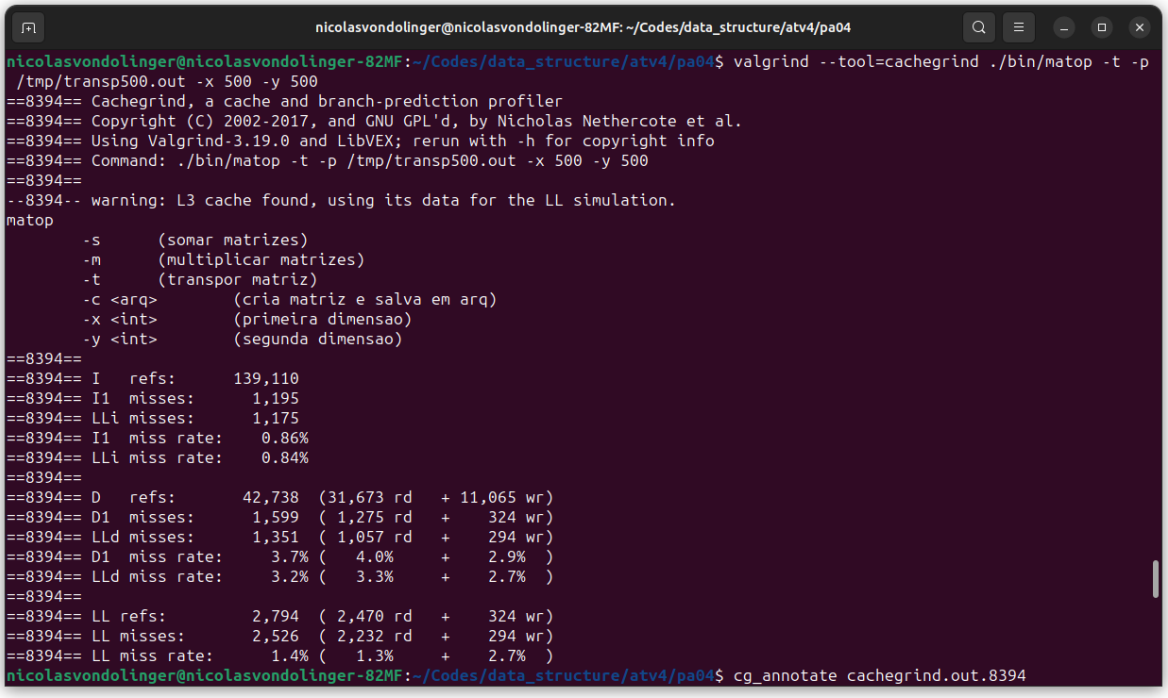


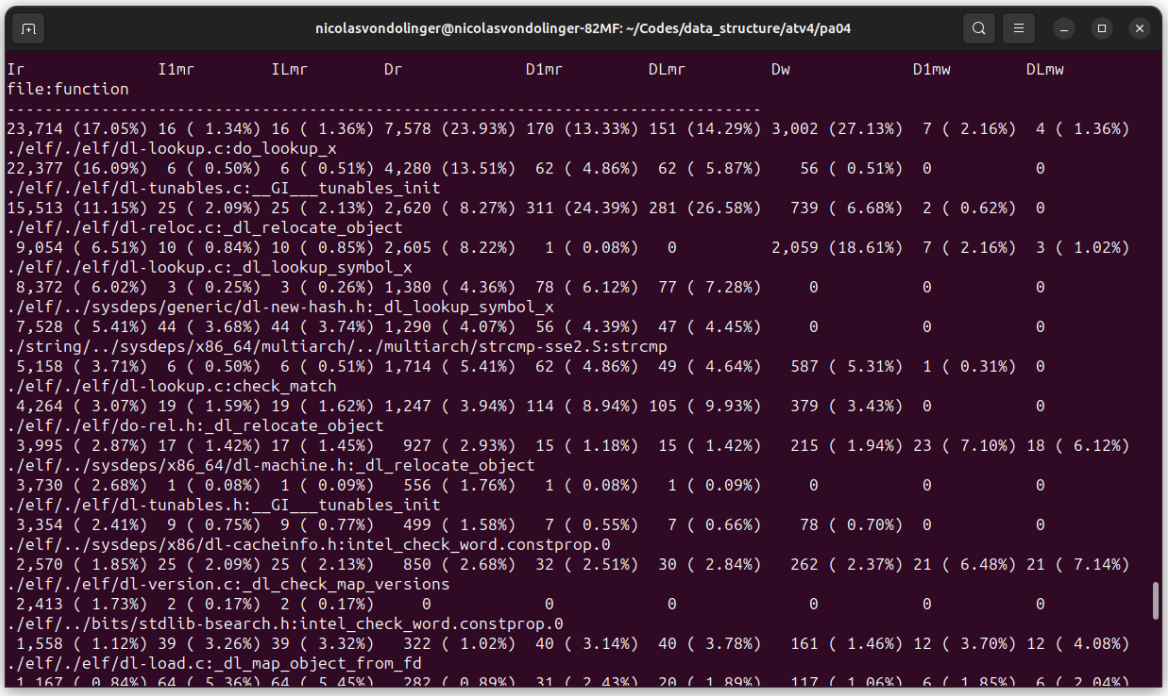
Multiplicação de Matrizes



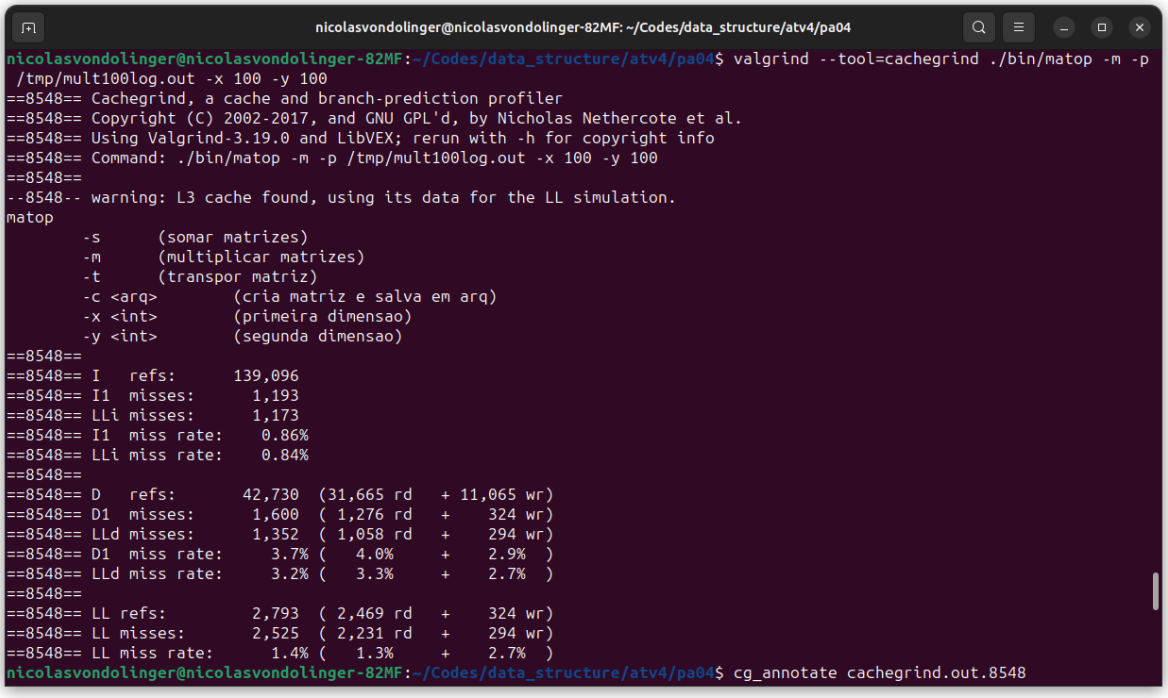


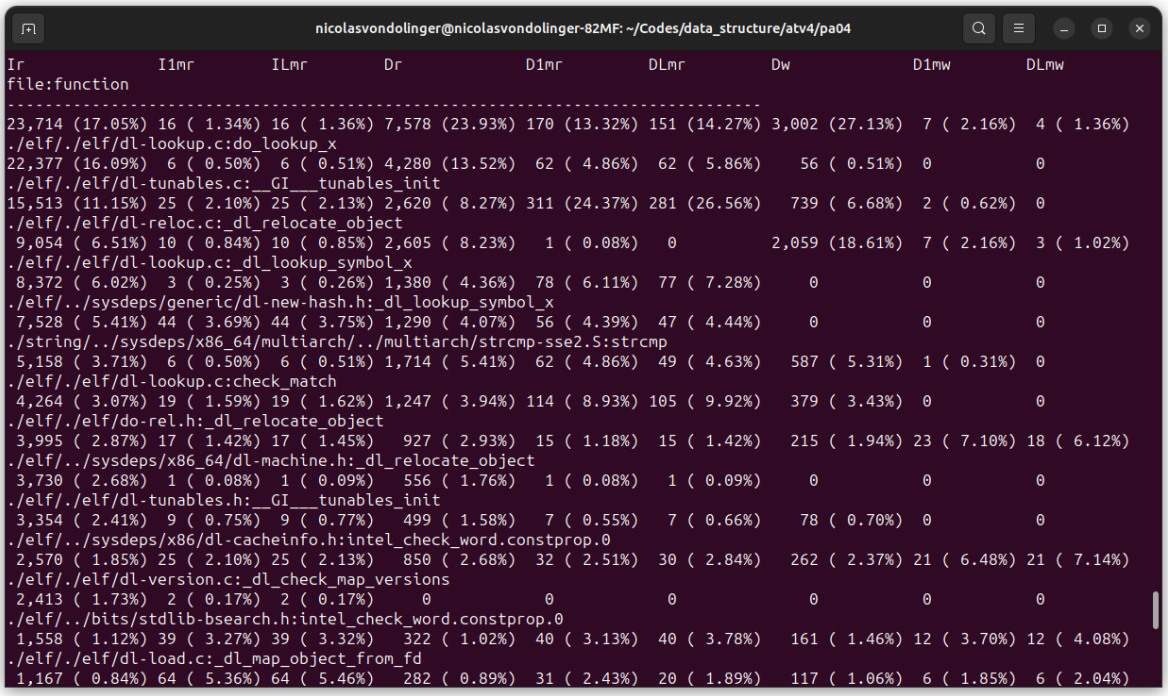
Transposição de Matrizes





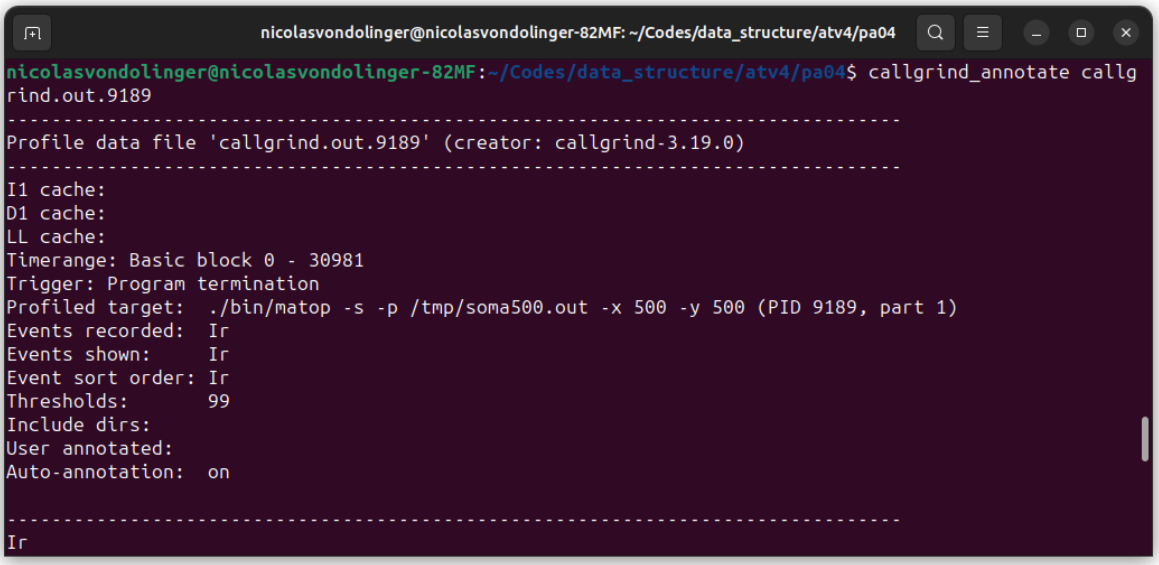
Multiplicação de Matrizes com Registro de Acesso



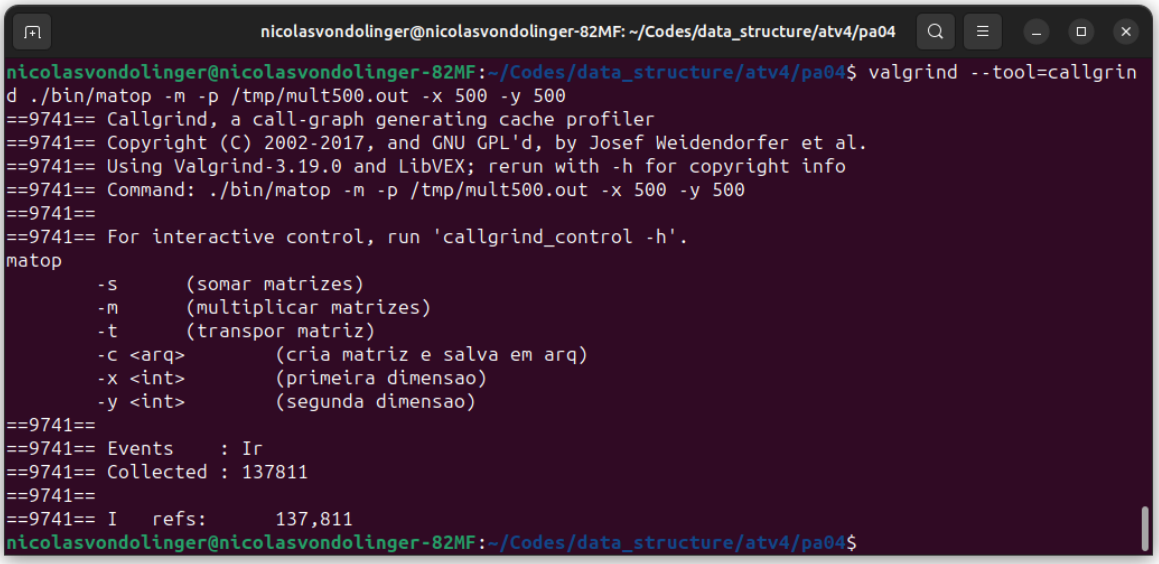


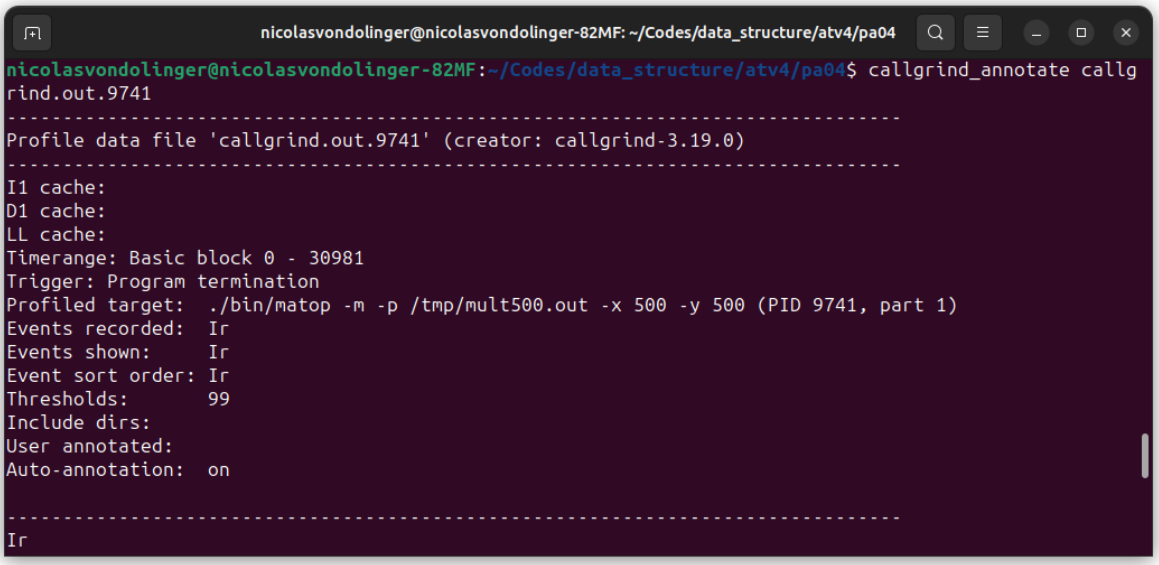
Callgrind

Soma de Matrizes

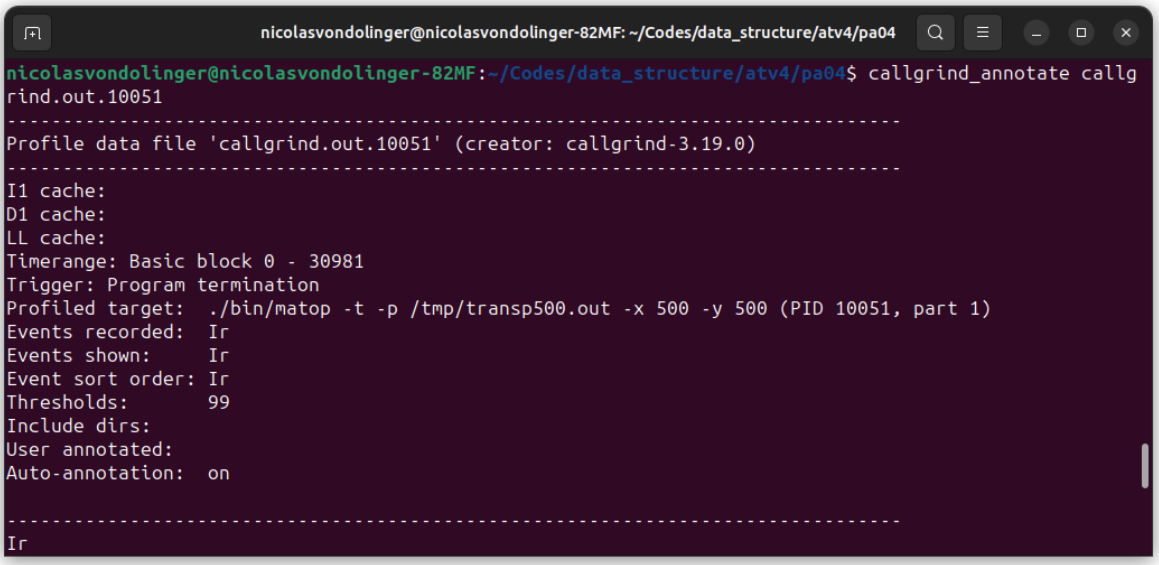


Multiplicação de Matrizes

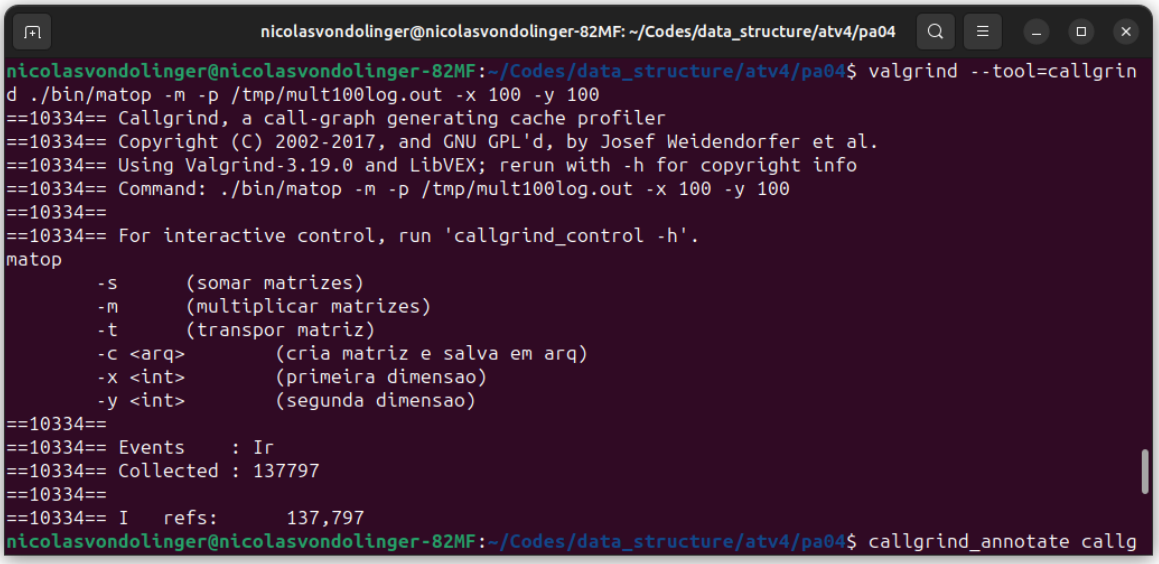


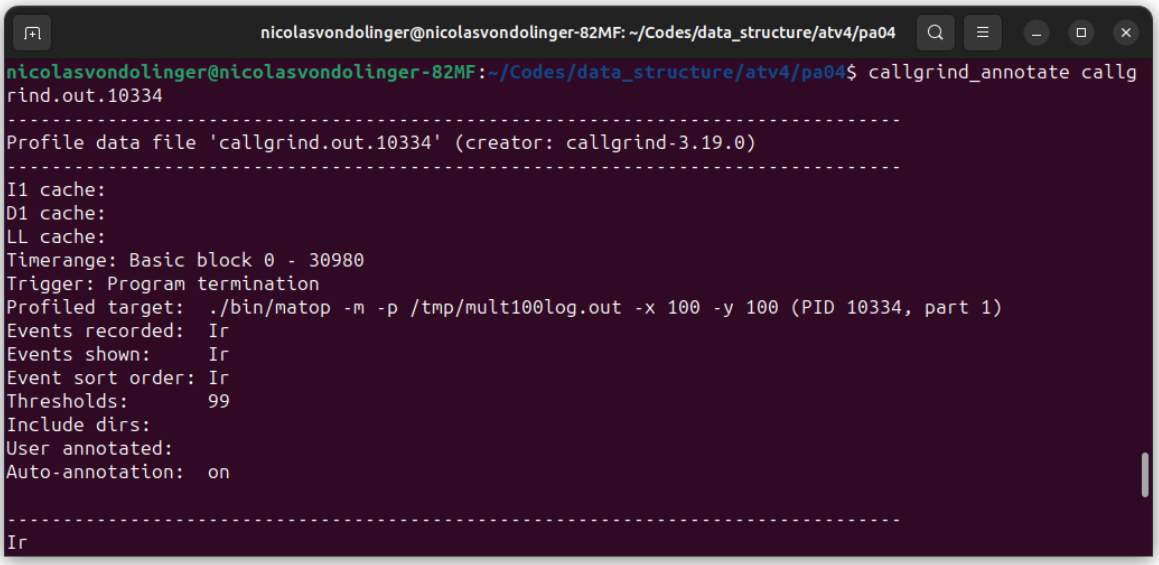


Transposição de Matrizes



Multiplicação de Matrizes com Registro de Acesso





Análise das Saídas

Cachegrind

1. Quão bem o programa se comporta em termos de memória?

   O relatório referente ao código mostra que a função do_lookup_x e __GI___tunables_init estão consumindo uma quantidade significativa de recursos. No geral, o programa executou 139.110 instruções (Ir) e teve 31.673 leituras de dados (Dr). A maioria dos eventos está associada a instruções de leitura (I1mr, ILmr) e leituras de dados (Dr, D1mr, DLmr).

   Essas funções são responsáveis por uma parcela substancial do tempo de execução do programa, com "do_lookup_x" representando 17,05% das instruções executadas (Ir) e "__GI___tunables_init" contribuindo com 16,09%. Outras funções, como "_dl_relocate_object" e "_dl_lookup_symbol_x," também são relevantes em termos de instruções executadas e leituras de dados. Com base nesses dados, é evidente que as funções "do_lookup_x" e "__GI___tunables_init" são as principais áreas a serem otimizadas em termos de consumo de recursos.

Em geral, com base nas informações fornecidas, o programa parece se comportar bem em termos de memória. A alta porcentagem de acertos na memória cache sugere que o programa está aproveitando eficientemente o uso da cache para reduzir os acessos à memória principal, o que é fundamental para o desempenho de muitas aplicações.

2. Quais estruturas de dados devem ser caracterizadas para melhor entendimento?

   Hierarquia de Cache: Entender a hierarquia de cache, incluindo o tamanho e a associatividade de cada nível de cache (L1, L2, L3, etc.), ajuda a determinar como os dados estão sendo armazenados e acessados em diferentes camadas de cache.

   Acessos à Memória: Analisar as métricas de leitura (read) e escrita (write) para cada nível de cache, bem como os acertos (hits) e falhas (misses), fornece informações sobre a eficiência do uso da memória cache. Isso inclui o número absoluto de acessos e a porcentagem de acertos em cada nível.

   Padrões de Acesso: Identificar os padrões de acesso à memória, como localidade espacial (acessos a dados próximos na memória) e localidade temporal (reutilização de dados recentemente acessados), é crucial para entender como o programa está acessando os dados.

   Funções Críticas: Analisar as estatísticas de acesso à memória para funções específicas no código-fonte pode ajudar a identificar quais partes do programa estão contribuindo mais para o tráfego de memória e onde podem ocorrer oportunidades de otimização.

   Tamanho dos Dados: Avaliar o tamanho dos dados usados pelo programa pode ajudar a determinar se os dados cabem eficientemente na memória cache disponível ou se há necessidade de estratégias de otimização, como particionamento de dados.

Padrões de Escrita: Entender como as operações de escrita afetam a memória cache e a memória principal é importante para otimizar o programa. Pode ser útil identificar padrões de escrita, como escrita única ou escrita múltipla em um mesmo local de memória.

Dependência de Dados: Identificar dependências de dados entre diferentes partes do programa pode ajudar a determinar a ordem de execução das operações e a minimizar os conflitos na memória cache.

Políticas de Substituição e Escrita: Conhecer as políticas de substituição de cache (por exemplo, LRU, FIFO) e as políticas de escrita (por exemplo, write-through, write-back) usadas pelo hardware de cache é importante para entender como os dados são gerenciados na memória cache.

Tamanho dos Blocos de Cache: Compreender o tamanho dos blocos de cache (também conhecidos como linhas de cache) é importante para entender como os dados são agrupados e carregados na memória cache.

Instruções vs. Dados: Distinguir entre acessos à memória para instruções e dados é fundamental, pois os padrões de acesso podem variar significativamente entre os dois tipos de acesso.

3. Quais segmentos de código devem ser instrumentados para suportar a caracterização?

   Para suportar a caracterização do desempenho de um programa, é importante instrumentar os segmentos de código que são críticos para a análise de cache e memória. Aqui estão alguns segmentos de código que devem ser instrumentados:

   Pontos de Entrada: como a função main() ou outras funções principais que iniciam a execução. Isso permite rastrear o comportamento do programa desde o início.

Funções Críticas: Isso pode incluir funções que realizam cálculos intensivos, operações de E/S (entrada e saída), ou funções que manipulam grandes volumes de dados.

Laços de Processamento: laços que iteram sobre estruturas de dados grandes.

Acessos à Memória: acessos à memória, incluindo leituras e escritas em variáveis e estruturas de dados. Isso ajudará a rastrear onde os dados estão sendo acessados e modificados.

Pontos Críticos de Cache: pontos críticos onde os dados são frequentemente carregados na memória cache e onde os acertos ou falhas de cache têm um impacto significativo no desempenho. Isso pode incluir operações de busca ou acesso a estruturas de dados compartilhadas.

Acesso a Estruturas de Dados Complexas: como matrizes multidimensionais, listas encadeadas ou árvores, que podem ter padrões de acesso à memória distintos.

Callgrind

1. Quão bem o programa se comporta em termos de memória?

   Com base apenas no perfil do Callgrind fornecido, não se pode tirar conclusões definitivas sobre o comportamento do programa em termos de memória. Para avaliar o uso de memória do programa, deveria ser realizada uma análise de perfil de memória usando as ferramentas apropriadas. No entanto, para todos os códigos de manipulação de matrizes, o número total de instruções retiradas é 137.811.

As estruturas de dados e os segmentos de código são os mesmos da análise feita do cachegrind.

Conclusão

O código fornecido apresenta uma complexidade de acesso à memória significativa devido às operações em matrizes e ao potencial uso de estruturas de dados complexas. A análise qualitativa revelou a presença esperada de localidade de referência espacial e temporal, bem como a identificação das estruturas de dados e segmentos de código críticos que podem afetar o desempenho.

A caracterização de localidade de referência é fundamental para a otimização do código, permitindo a identificação de oportunidades de melhoria no acesso à memória e, consequentemente, no desempenho geral. Este relatório fornece uma visão geral inicial da complexidade de acesso à memória no código e serve como ponto de partida para análises mais aprofundadas e otimizações futuras.