Relatório da Tarefa 02

Pipelines e Vetorização

20210093987 - Francisca Paula de Souza Braz
Data da entrega: 03/04/2025
Disciplina: DCA3703 - PROGRAMAÇÃO PARALELA - T01 (2025.1)

Introdução

A vetorização e o pipeline trabalham juntos para melhorar o desempenho, a **vetorização** permite que múltiplas operações sejam realizadas simultaneamente dentro de um mesmo ciclo. E o **pipeline** melhora a eficiência da execução, garantindo que diferentes etapas das instruções estejam sempre em andamento. Quando combinadas, essas técnicas possibilitam uma execução mais rápida e eficiente, reduzindo gargalos e maximizando o uso dos recursos do processador.

Nesta atividade, implementamos três versões de um laço em C para analisar os efeitos do **pipeline** e da **vetorização** no tempo de execução de um programa. A primeira versão inicializa um vetor e realiza a soma de seus elementos de forma sequencial, criando uma dependência entre as iterações, o que pode limitar o aproveitamento do pipeline. A segunda versão tenta minimizar essa dependência ao utilizar múltiplas variáveis acumuladoras, permitindo que o processador execute mais instruções em paralelo, maximizando o uso do pipeline.

Para avaliar o impacto dessas técnicas, compilamos e executamos o código com diferentes níveis de otimização (-O0, -O1, -O2 e -O3), medindo o tempo de execução em cada caso. Os resultados obtidos mostram como o estilo do código e as otimizações do compilador afetam o desempenho, permitindo um melhor aproveitamento da arquitetura do processador para reduzir o tempo de execução.

Metodologia

Implementação do Código

Para investigar os efeitos do **pipeline** e da **vetorização** no desempenho do código, desenvolvemos um programa em C que realiza três operações principais:

- Inicialização de um vetor com **`TAMANHO = 1.000.000`**, preenchendo-o com valores sequenciais.
- Soma acumulativa sequencial, onde cada iteração depende do valor calculado na iteração anterior, o que pode limitar o aproveitamento do pipeline.

- Soma acumulativa otimizada, utilizando duas variáveis acumuladoras para minimizar dependências entre iterações, permitindo que o processador explore melhor o pipeline e execute mais instruções simultaneamente.
- A medição do tempo de execução para essa atividade também foi feita utilizando a função `clock_gettime(CLOCK_MONOTONIC, &tempo)`, registrando o tempo antes e depois da execução de cada laço.

Compilação e Execução

O código foi compilado com o GCC (GNU Compiler Collection) utilizando diferentes níveis de otimização: `-O0` (sem otimização), `-O2` (otimizações avançadas, incluindo vetorização), `-O3` (otimizações agressivas, explorando ao máximo o pipeline e a vetorização) e Cada versão foi executada e os tempos de execução foram coletados para análise.

Teste de Vetorização

Para entender melhor a vetorização realizada pelos otimizadores e verificar se o compilador vetorizou realmente o código, utilizamos o seguinte comando:

```
gcc -O3 -ftree-vectorize -fopt-info-vec-optimized -o atividade_2 atividade_2.c
```

Após a execução, **nenhuma mensagem indicando vetorização foi exibida**, sugerindo que a soma acumulativa **não foi vetorizada automaticamente** pelo compilador.

Resultados

A tabela abaixo apresenta os tempos obtidos para cada operação com diferentes níveis de otimização:

Otimização	Inicialização (s)	Soma Acumulativa 1 (s)	Soma Acumulativa 2 (s)
-	0.002055	0.002108	0.001178

-O0	0.002216	0.002512	0.001704
-O2	0.000523	0.000649	0.000471
-O3	0.000294	0.000432	0.000372

Comparando a execução sem otimizações “-O0” com “-O3”, o tempo da soma acumulativa caiu de 0.002512s para 0.000432s, uma redução de aproximadamente **83%** com isso podemos observar que as otimizações reduziram significativamente o tempo de execução.
Em todos os casos, a segunda versão da soma (com múltiplas variáveis) obteve um desempenho melhor do que a versão sequencial, pois quebrou as dependências entre as iterações e possibilitou melhor aproveitamento do pipeline e com isso notou-se que a soma acumulativa otimizada foi mais eficiente.

Como esperado, a otimização “-O3” resultou nos menores tempos, aplicando técnicas avançadas como reordenação de instruções, melhor aproveitamento do pipeline e, possivelmente, vetorização automática, com isso podemos afirmar que para a atividade a otimização “-O3” apresentou o melhor desempenho.

Com as otimizações “-O2” e “-O3”, o compilador ativou técnicas de vetorização, permitindo que múltiplas operações fossem realizadas em paralelo dentro de um único ciclo de CPU. O uso de múltiplas variáveis na segunda soma ajudou o pipeline do processador, reduzindo gargalos causados por dependências de dados e permitindo maior paralelismo na execução. Os resultados mostram como a organização do código e as otimizações do compilador impactam diretamente o desempenho, explorando ao máximo os recursos do hardware.
Após a realização das interações ainda rodei para a “-O3" um teste de vetorização, e como nenhuma mensagem foi exibida pode inferir que não foi vetorizado, dito isso algumas coisas poderiam ter acontecido devido a minha estrutura de código. A primeira soma acumulativa possui dependência entre iterações (**`soma += vetor[i]`**), impedindo a computação paralela de múltiplas iterações, e a segunda soma

acumulativa, apesar de quebrar parcialmente as dependências, ainda não permitiu vetorização total devido ao acesso indireto às variáveis (**`soma1 e soma2`**).

O compilador pode ter identificado que a vetorização não traria ganhos significativos para esse padrão de laço. Mesmo sem vetorização automática, os tempos de execução foram reduzidos devido à reordenação de instruções e ao melhor uso do pipeline da CPU. O compilador conseguiu minimizar gargalos causados por dependências de dados, permitindo que mais operações fossem executadas simultaneamente dentro dos ciclos do processador.

Conclusão

Os experimentos realizados demonstraram a importância da otimização de código e do aproveitamento eficiente dos recursos do processador, como o pipeline de instruções e a vetorização, para melhorar o desempenho de laços iterativos.

Observamos que a execução sem otimizações “-O0” resultou nos tempos mais altos, enquanto as otimizações mais agressivas "-O3" reduziram drasticamente o tempo de execução. Isso ocorreu devido à reordenação de instruções, ao melhor aproveitamento do pipeline e, em alguns casos, à aplicação da vetorização automática.

Além disso, a modificação na soma acumulativa, utilizando múltiplas variáveis acumuladoras, ajudou a minimizar dependências entre iterações. Essa abordagem permitiu que o compilador aplicasse técnicas de otimização mais eficientes, reduzindo os gargalos no pipeline e melhorando significativamente o desempenho. Com isso podemos concluir que o uso de otimizações do compilador tem um impacto significativo no desempenho do código, a quebra de dependências dentro de um laço permite melhor aproveitamento do pipeline, facilitando a execução paralela de instruções e Estratégias como o uso de múltiplas variáveis acumuladoras são eficazes para melhorar o desempenho de operações matemáticas intensivas.