Programação Paralela
Tarefa 2 - Pipeline e Vetorização

Anelma Silva da Costa
UFRN - DCA
anelma.costa.100@ufrn.edu.br

I - Introdução

Nos sistemas computacionais modernos, a otimização de código é essencial para maximizar o desempenho, especialmente em aplicações que demandam processamento intensivo, como simulações numéricas, machine learning e processamento de sinais. Duas técnicas fundamentais para melhorar a eficiência de execução são o Paralelismo ao Nível de Instrução (ILP - Instruction-Level Parallelism) e a vetorização, que permitem que o processador execute múltiplas operações simultaneamente, reduzindo o tempo total de processamento.

Este relatório investiga os efeitos do ILP por meio da implementação de três laços em C, cada um com características distintas:

Inicialização de um vetor com um cálculo simples, onde cada iteração é independente, permitindo alto grau de paralelismo.

Soma aculmulativa com dependência de dados, onde cada iteração depende do resultado da anterior, limitando o ILP.

Quebra de dependência usando múltiplas variáveis, uma técnica que permite ao compilador e ao hardware explorar melhor o paralelismo.

Os experimentos serão compilados com diferentes níveis de otimização (-O0, -O2, -O3) para avaliar como o compilador gcc transforma o código e como as dependências afetam o desempenho. A análise comparativa dos tempos de execução fornecerá insights sobre como estruturar algoritmos para melhor aproveitamento do ILP e da vetorização.

II - Teoria

- **Paralelismo ao Nível de Instrução (ILP)**

O ILP é uma técnica que permite a execução simultânea de múltiplas instruções em um único fluxo de execução (thread), explorando paralelismo intrínseco ao código. Processadores modernos utilizam várias estratégias para implementar ILP, incluindo:

- Pipelining: Divisão da execução de instruções em estágios (fetch, decode, execute, etc.), permitindo que várias instruções sejam processadas em paralelo em diferentes fases.

- Execução Superescalar: Capacidade de despachar múltiplas instruções por ciclo de clock, desde que não haja dependências entre elas.

- Especulação: Execução antecipada de instruções, mesmo quando há desvios condicionais, reduzindo tempos de espera.

No entanto, dependências de dados (quando uma instrução necessita do resultado de outra) podem limitar drasticamente o ILP. Por exemplo, em um laço de soma acumulativa (soma += vetor[i]), cada iteração depende da anterior, impedindo a paralelização.

- **Dependências de Dados e Técnicas de Otimização**

As dependências podem ser classificadas em:

- Verdadeiras (RAW - Read After Write): Quando uma instrução lê um valor que depende de uma escrita anterior.

- Antidependências (WAR - Write After Read): Quando uma escrita ocorre após uma leitura do mesmo dado.

- Dependências de Saída (WAW - Write After Write): Quando duas escritas ocorrem na mesma posição em ordem não determinística.

Para reduzir o impacto dessas dependências, técnicas como desenrolamento de laços (loop unrolling) e acumulação em múltiplos registradores podem ser aplicadas. Por exemplo, em vez de acumular a soma em uma única variável, usar múltiplos acumuladores permite que o compilador e o processador executem operações em paralelo.

- **Efeito dos Níveis de Otimização do Compilador**

Compiladores como GCC e Clang aplicam transformações agressivas no código, dependendo do nível de otimização:

- -O0 (Sem otimização): Mantém a estrutura original do código, sem ILP significativo.

- -O2 (Otimização padrão): Aplica pipelining, desenrolamento de laços e reordenação de instruções.

- -O3 (Otimização máxima): Inclui vetorização (SIMD) e transformações mais arrojadas para extrair ILP.

A eficácia dessas otimizações depende da ausência de dependências rígidas no código.

- **Vetorização e Instruções SIMD**

Além do ILP, compiladores podem converter laços em operações vetoriais usando instruções SIMD (Single Instruction, Multiple Data), como AVX (Intel) ou NEON (ARM). Isso é particularmente eficaz em laços independentes, onde operações podem ser aplicadas a múltiplos elementos simultaneamente.

III - Aplicação Prática

Para cumprir o objetivo da tarefa foi desenvolvido um código em C, na qual para o primeiro laço (Cálculo simples) foi construído uma soma simples, do valor do vetor como número 2. Já no segundo laço, temos uma soma acumulativa com dependência, na qual o valor atual a ser somado depende do valor anterior do vetor.

O último laço também é uma soma acumulativa, porém sem dependência do valor anterior. Neste caso há uma quebra da dependência ao utilizar mais de uma variável para realizar a soma. Abaixo é possível verificar cada um dos laços, na ordem que foram listados aqui.

```c
// 1º Laço: Calculo Simples
start = get_time();
for (int i = 0; i < N; i++) {
    A[i] = i + 2;
}
end = get_time();
printf("Tempo do calculo simples foi de: %.6f segundos\n", end - start);

// 2º Laço: Soma acumulativa (dependência entre iterações)
start = get_time();
soma = 0;
for (int i = 0; i < N; i++) {
    soma += A[i];
}
end = get_time();
printf("\nTempo Soma acumulativa dependente: %.6f segundos\n", end - start);
printf("Soma: %d\n", soma);

// 3º Laço: Soma com quebra de dependência
start = get_time();
soma1 = 0;
soma2 = 0;
for (int i = 0; i < N; i += 2) {
    soma1 += A[i];
    soma2 += A[i + 1];
}
soma = soma1 + soma2;
end = get_time();
printf("\nTempo Soma acumulativa independente: %.6f segundos\n", end - start);

printf("Soma final: %d\n", soma);
```

- **Laços x Pipeline e Vetorização**

- **Laço do cálculo simples:** O pipeline se beneficia das iterações que são independentes, o processador pode iniciar a execução de uma nova iteração enquanto a anterior ainda está em andamento em outra parte do pipeline, aproveitando melhor os ciclos de clock disponíveis. Com a vetorização, o processador pode usar uma única instrução para realizar múltiplas operações em paralelo. Isso é especialmente benéfico para laços como o primeiro, onde não há dependências entre as iterações e as operações são simples.

- **Laço da soma acumulativa (com dependência):** Como o valor de soma é atualizado a cada iteração e essas atualizações dependem da iteração anterior, o processador não pode facilmente iniciar a execução da próxima iteração antes que a soma da iteração anterior tenha sido concluída. Isso cria ineficiências no pipeline, pois o

processador precisa aguardar a conclusão da atualização da variável soma antes de passar para a próxima iteração. Já na vetorização, embora o compilador possa tentar otimizar o laço de várias maneiras (como unrolling ou utilizando algumas instruções vetoriais), o fato de que a operação sobre soma é dependente entre iterações impede a vetorização completa.

- **Laço da soma acumulativa (sem dependência):** Como as operações sobre soma1 e soma2 são independentes, o processador pode executar essas operações em paralelo (ou em pipelining de forma mais eficiente), o que aproveita melhor o pipeline. Isso pode reduzir o número de ciclos necessários para concluir o laço, pois não há dependência de dados entre as duas somas. O processador pode pipelinear cada operação de forma separada, melhorando a eficiência do uso de recursos. Na vetorização o compilador pode usar instruções vetoriais para somar múltiplos elementos de A[i] ao mesmo tempo. Isso pode acelerar significativamente o laço, pois em vez de processar um valor de A[i] de cada vez, o processador pode operar sobre vários valores simultaneamente. Esse tipo de operação se torna ainda mais eficiente com otimização de paralelismo em sistemas com múltiplos núcleos.

    - **Laço x Otimizações**

- **Laço do cálculo simples:** A otimização geralmente resulta em uma melhoria visível aqui, uma vez que o laço é simples e pode ser melhorado pelo compilador para aproveitar o pipeline de execução do processador.

- **Laço da soma acumulativa (com dependência) :** Mesmo com otimizações, o desempenho não se beneficiará tanto quanto os outros laços, pois a dependência de dados entre as iterações impede otimizações como paralelização.

- **Laço da soma acumulativa (sem dependência) :** Este laço tende a se beneficiar bastante de otimizações, especialmente quando o código é compilado com níveis de otimização mais altos. O fato de as somas de soma1 e soma2 serem independentes pode ser explorado em paralelo.

-

O código completo do tarefa pode ser encontrado através do link: https://github.com/AnelmaSilva/Programa-ao_Paralela_25_1/tree/main/Tarefa%202 .

IV - Teoria x Resultados Práticos

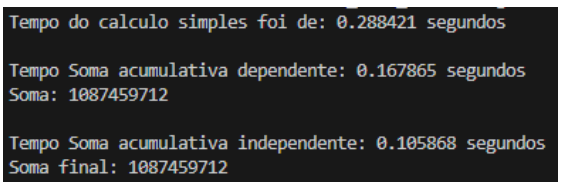
Dados o conteúdo visto até aqui, de forma teórica podemos esperar que o terceiro laço seja mais rápido que o segundo laço, especialmente em altos níveis de otimização. Já o primeiro laço tende a ser beneficiado pela vetorização em altos níveis de otimização.

Para obtenção dos resultados práticos, o código foi desenvolvido e executado na seguinte ordem:

- Um vetor de tamanho N = 100000000 foi utilizado para todas as versões de laços.
- Primeiro foi executado utilizando um nível de otimização -O0.
- Em segundo foi executado utilizando um nível de otimização -O2.
- E por último foi executado utilizando um nível de otimização -O3.

- Resultados

Otimização -O0:

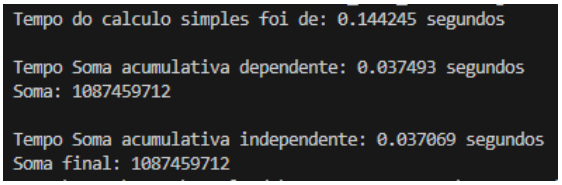


Primeiro laço: operações independentes, mas mais lento devido à escritas em memória.

Segundo laço: Mais rápido que o primeiro laço (leituras são mais eficientes que escritas), mas limitado pela dependência sequencial (soma += A[i]).

Terceiro laço: 37% mais rápido que o laço dependente. Paralelismo implícito ao dividir a soma em soma1 e soma2, reduzindo gargalos.

Otimização -O2:

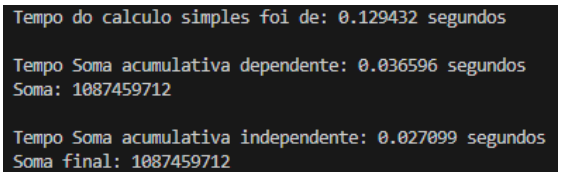


Primeiro laço: 50% mais rápido que -O0. Melhoria significativa devido a otimizações de loop (possivelmente unrolling), melhor aproveitamento do pipeline do processador, possível vetorização das operações de escrita.

Segundo laço: 78% mais rápido que -O0. Apesar da dependência, o compilador conseguiu reordenar instruções para melhorar o ILP, otimizar o acesso à memória e aplicar técnicas de especulação.

Terceiro laço: 65% mais rápido que -O0. Performance similar à versão dependente, indicando que as otimizações do -O2 foram tão eficientes para o caso dependente, possível vetorização em ambos os casos e overhead de gerenciar múltiplos acumuladores pode ter compensado os benefícios.

Otimização -O3:



Primeiro laço: 10,3% mais rápido que o -O2, porém mais lento em todas as otimizações.

Segundo laço: 2.4% mais rápido que o -O2, quebra da dependência com técnicas como unrolling porém com ganho mínimo, assim como na vetorização

Terceiro laço: 26.9% mais rápido que o -O3, melhor exploração da ILP mantendo as duas acumulações totalmente independentes, uso de instruções SIMD para somar pares de elementos, unrolling mais agressivo.

V - Conclusões

Os resultados obtidos demonstram claramente a importância da otimização de código e do aproveitamento do Paralelismo ao Nível de Instrução (ILP) e da vetorização (SIMD) em processadores modernos. A análise comparativa entre os diferentes níveis de otimização (-O0, -O2, -O3) revelou que:

Laço Independente (Cálculo Simples): Beneficiou-se significativamente das otimizações de loop (unrolling) e vetorização, especialmente em -O3, onde operações de escrita foram parcialmente paralelizadas. Ainda assim, foi o mais lento devido ao custo inerente de acessos à memória.

Laço Dependente (Soma Acumulativa Sequencial): Teve desempenho limitado pela dependência de dados (RAW), mesmo com otimizações agressivas.

-O2 e -O3 melhoraram o desempenho via pipelining e reordenação de instruções, mas o gargalo persistiu.

Laço com Quebra de Dependência (Soma Paralelizável): Obteve o melhor desempenho em -O3, com ganho de ~27% em relação a -O2, graças à exploração eficiente de ILP e vetorização. A divisão em acumuladores independentes permitiu que o compilador e o hardware executassem operações em paralelo, maximizando a utilização do pipeline e de instruções SIMD.

Portanto, para maximizar desempenho em aplicações intensivas, reduzir dependências e habilitar otimizações avançadas são estratégias essenciais.

VI - Referências

- Livro: An introduction to parallel programming .
- Conteúdo ministrado em sala de aula.
- IA's (DeepSeek e ChatGPT)