UNIVERSIDADE ESTADUAL DE SANTA CRUZ

BEATRIZ SANTOS DE OLIVEIRA E JÚLIA DE ARAÚJO RAMOS

Processamento paralelo: Relatório de comparação entre multiplicação de matrizes (DGEMM) usando metódo sequencial e processamento paralelo com OpenMP

ILHÉUS - BAHIA

2025

BEATRIZ SANTOS DE OLIVEIRA E JÚLIA DE ARAÚJO RAMOS

Processamento paralelo: Relatório de comparação entre multiplicação de matrizes (DGEMM) usando metódo sequencial e processamento paralelo com OpenMP

Trabalho avaliativo da disciplina Processamento paralelo ministrada pelo professor Esbel Tomás Valero Ollerana.

ILHÉUS - BAHIA
2025

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

O presente relatório detalha o desenvolvimento e a análise de desempenho de um projeto focado na aplicação de técnicas de paralelismo em arquiteturas de memória compartilhada. O objetivo central do trabalho foi implementar e comparar duas versões de uma rotina de multiplicação de matrizes de precisão dupla (DGEMM): uma puramente sequencial e outra paralela, otimizada com o uso de diretivas da API OpenMP. Através da experimentação com matrizes de dimensões crescentes (512x512 a 4096x4096) e da variação no número de threads, busca-se quantificar e analisar os ganhos de desempenho obtidos com a computação paralela nesta operação da álgebra linear.

Para a correta compreensão do projeto, é importante revisitar os conceitos teóricos que o fundamentam. O primeiro é a multiplicação de matrizes, uma operação que calcula cada elemento da matriz resultante a partir do produto escalar entre as linhas da primeira matriz e as colunas da segunda. Sua complexidade computacional, na ordem de $O(n^3)$, a torna um gargalo em muitas aplicações e, portanto, uma candidata ideal à otimização (Cormen et al. 2009). Para mitigar esse custo, emprega-se o paralelismo, uma técnica que consiste em dividir uma tarefa computacional em subtarefas que podem ser executadas simultaneamente por múltiplos núcleos de processamento (Grama et al. 2003). Por fim, para implementar essa estratégia em um ambiente de memória compartilhada, utilizou-se o OpenMP, uma API que simplifica a programação paralela através de diretivas de compilador (ex: #pragma omp parallel), instruindo o compilador a distribuir o trabalho entre diferentes threads e a gerenciar a sua execução e sincronização (Chapman et al. 2021).

A simples implementação de um código paralelo, no entanto, não garante uma melhoria de desempenho. Medir apenas o tempo de execução não é suficiente para compreender a fundo os resultados obtidos. Métricas como o Speedup, que quantifica o ganho de velocidade da versão paralela em relação à sequencial, são fundamentais para uma avaliação crítica (Hennessy e Patterson 2019). Essa análise permite não apenas validar a eficácia da paralelização, mas também identificar gargalos, overheads de comunicação e sincronização, e os limites de escalabilidade da solução proposta, oferecendo uma visão completa sobre os benefícios e os custos da transição de um algoritmo sequencial para um paralelo (Amdahl 1967).

2 Metodologia

Nesta seção, são detalhados os procedimentos e as ferramentas utilizadas para o desenvolvimento, execução e análise das implementações sequencial e paralela da rotina DGEMM.

2.1 Descrição da Implementação Sequencial

A versão sequencial foi escrita em linguagem C e serve como base de comparação para medir o desempenho das versões otimizadas. Nela, foi implementada a função dgemm, que realiza a multiplicação de matrizes. Para aproveitar melhor a hierarquia de memória do processador, as matrizes, que normalmente são pensadas como bidimensionais, foram guardadas em forma de vetores unidimensionais (ou arranjos lineares). Isso garante que os dados fiquem armazenados de maneira contínua na memória, o que facilita o acesso e torna o processamento mais rápido. Por exemplo, em vez de acessar um elemento como $A[i][j]$, ele passa a ser acessado como $A[i * n + j]$, onde n é o tamanho da matriz.

A multiplicação das matrizes foi feita usando a ordem de laços $i - k - j$. Nesse esquema, o laço de fora percorre as linhas da matriz A (índice i), o laço do meio percorre as colunas de A e as linhas de B (índice k), e o laço de dentro percorre as colunas de B (índice j). Essa ordem é vantajosa porque o valor $A[i * n + k]$ pode ser carregado uma vez em um registrador do processador e reutilizado em todas as iterações do laço mais interno. Isso melhora a localidade de dados e evita que o mesmo valor precise ser buscado várias vezes na memória.

Além disso, foi aplicada uma técnica de otimização chamada Loop Unrolling no laço mais interno (o j). Essa técnica consiste em expandir manualmente o corpo do laço para que, a cada iteração, em vez de calcular apenas um elemento da matriz resultante C, sejam calculados quatro elementos de uma vez. Com isso, reduz-se a quantidade de verificações de condição e saltos de laço (que também consomem tempo de CPU) e o compilador pode, por exemplo, em vez de rodar um laço for que soma elemento por elemento, "desenrolar"esse laço e fazer várias somas seguidas dentro da mesma iteração, diminuindo a sobrecarga e acelerando o cálculo.

2.2 Descrição da Implementação Paralela (OpenMP)

A implementação paralela foi feita a partir da versão sequencial otimizada, mas adicionando diretivas do OpenMP para aproveitar o uso de várias threads ao mesmo

tempo. A principal delas foi o #pragma omp parallel for, colocada no laço mais externo (o i, que percorre as linhas das matrizes A e C). Isso significa que cada thread vai cuidar de calcular um conjunto de linhas inteiras da matriz. Como cada linha pode ser calculada de forma independente, não há risco de conflito entre threads e também não é preciso usar sincronização extra. Além disso, essa escolha é eficiente porque as threads só são criadas uma vez no início e sincronizadas no final.

Depois, foi usado o schedule(static). Esse agendamento divide as linhas entre as threads de forma fixa e equilibrada, antes mesmo do cálculo começar. Isso funciona muito bem aqui porque todas as linhas têm o mesmo custo de processamento, ou seja, o trabalho já é naturalmente balanceado. O static ainda é a opção mais rápida porque tem pouca sobrecarga.

Por fim, a diretiva #pragma omp simd foi inserida imediatamente antes do laço interno (j), que foi manualmente desdobrado (unrolled). A diretiva simd é uma instrução para o compilador, informando que o laço seguinte não possui dependências entre iterações e pode ser vetorizado com segurança. Isso permite que o processador utilize instruções SIMD (Single Instruction, Multiple Data), que realizam a mesma operação (neste caso, soma e multiplicação) em múltiplos dados simultaneamente, explorando o paralelismo em nível de instrução e complementando o paralelismo em nível de thread da diretiva principal.

2.3 Descrição do Hardware Utilizado

Todos os testes foram compilados e executados no mesmo ambiente de hardware e software para garantir a consistência e a comparabilidade dos resultados. As especificações do sistema são: processador AMD Ryzen 5 5600 (6 núcleos, 12 threads), memória RAM de 16 GB DDR4, Windows 11 e compilador GCC 15.2.0.

2.4 Procedimentos para Medição de Tempo

A medição de tempo foi realizada de maneira distinta para cada versão, utilizando as funções mais apropriadas para cada contexto. Na versão sequencial, o tempo foi medido utilizando a função clock() da biblioteca <time.h>. Esta função mede o tempo de CPU consumido pelo processo, o que é uma métrica adequada para um programa de thread única. Já na versão paralela, o tempo foi medido utilizando a função omp_get_wtime() da biblioteca <omp.h>. Esta função retorna o tempo de parede (wall-clock time), que é o tempo real decorrido entre dois pontos. Ela captura o tempo total de execução do ponto de vista do usuário, incluindo o tempo em que as threads podem estar ociosas ou em sincronização.

2.5 Métricas Utilizadas para Avaliação

Para realizar a análise de desempenho, foram selecionadas as seguintes métricas:

- **Tempo de Execução (s):** A métrica primária, medida em segundos, que representa o tempo total para executar a função `dgemm`.
- **Speedup:** Indica o ganho de desempenho da versão paralela em relação à sequencial. Foi calculado como:

$$Speedup(p) = \frac{\text{Tempo Sequencial}}{\text{Tempo Paralelo}(p)}$$

em que p é o número de threads utilizadas.

3 Resultados

Nesta seção, são apresentados e analisados os dados de desempenho coletados durante a execução das versões sequencial e paralela do algoritmo DGEMM. Os resultados foram consolidados em uma tabela e gráficos para facilitar a comparação, seguida de uma análise sobre a escalabilidade e speedup da solução paralela.

3.1 Tabela de Desempenho

Para garantir a fidedignidade dos dados, cada configuração foi executada três vezes, e o tempo de execução apresentado na Tabela 1 corresponde à média aritmética dessas execuções. O tempo da versão sequencial serviu como linha de base para o cálculo do speedup da versão paralela, que foi testada com 2, 4, 6, 8, 10 e 12 threads.

Tabela 1 – Resultados consolidados de tempo de execução e Speedup.

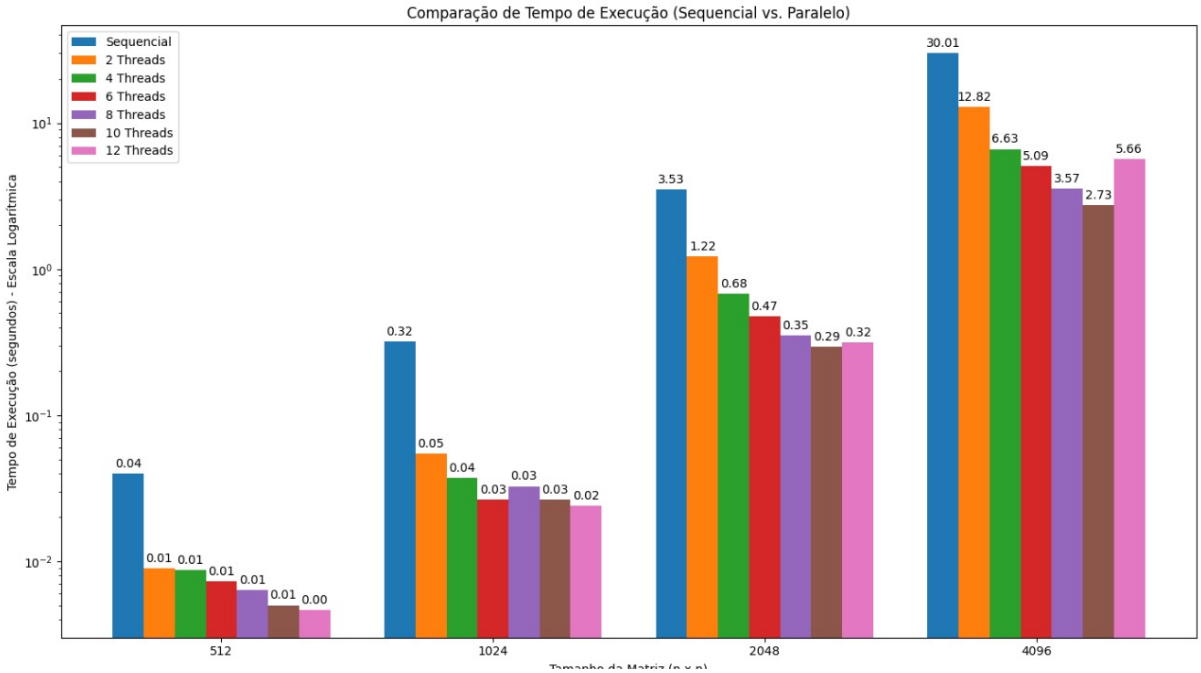
Tamanho (N)	Threads (p)	Tempo Médio (s)	Speedup
512	**1 (Seq.)**	**0.0397**	**-**
	2	0.0090	4.41x
	4	0.0090	4.41x
	6	0.0077	5.17x
	8	0.0063	6.28x
	10	0.0050	7.94x
	12	0.0047	8.51x
1024	**1 (Seq.)**	**0.3177**	**-**
	2	0.0547	5.81x
	4	0.0303	10.47x
	6	0.0300	10.59x
	8	0.0327	9.72x
	10	0.0263	12.08x
	12	0.0240	13.24x
2048	**1 (Seq.)**	**3.5260**	**-**
	2	1.3890	2.54x
	4	0.6977	5.05x
	6	0.4937	7.14x
	8	0.3497	10.08x
	10	0.2940	11.99x
	12	0.3157	11.17x
4096	**1 (Seq.)**	**30.0097**	**-**
	2	13.0263	2.30x
	4	6.4800	4.63x
	6	4.8993	6.13x
	8	3.5727	8.40x
	10	2.7283	11.00x
	12	5.6577	5.30x

3.2 Análise Gráfica dos Resultados

Para complementar a análise tabular, foram gerados gráficos que permitem uma visualização mais intuitiva do desempenho do algoritmo. A seguir, são apresentados e discutidos os gráficos de tempo de execução e de escalabilidade.

3.2.1 Comparação de Tempo de Execução

O gráfico de barras na Figura 1 compara diretamente o tempo de execução da versão sequencial com as versões paralelas para cada tamanho de matriz. A utilização de uma escala logarítmica no eixo vertical serve para visualizar a grande disparidade de performance entre as execuções.



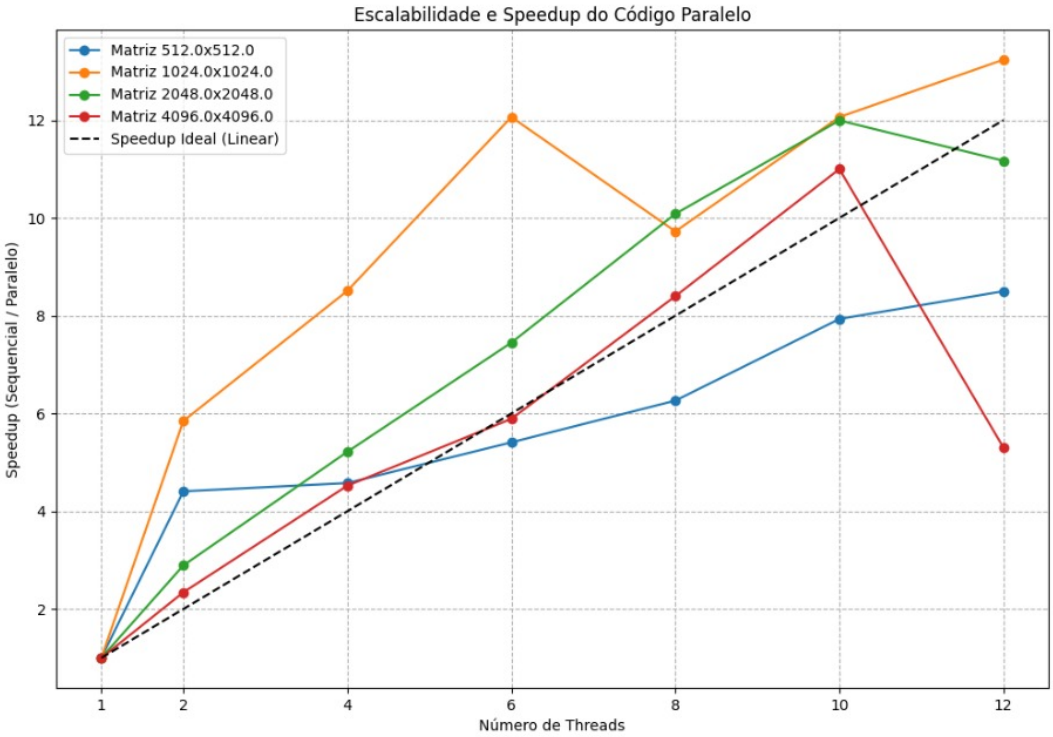
Fonte: Autoria própria.

Figura 1 – Comparação do tempo de execução (em segundos) em escala logarítmica.

A análise visual do gráfico reforça as conclusões da tabela. Observa-se uma redução drástica do tempo: para todos os tamanhos de matriz, há uma queda na altura das barras à medida que o número de threads aumenta, demonstrando o sucesso da paralelização até um certo ponto. Também é possível perceber o impacto do tamanho do problema, já que fica evidente o crescimento cúbico ($O(n^3)$) do tempo de execução. Por fim, nota-se o fenômeno dos retornos decrescentes, que agora resulta em uma diminuição do desempenho. É perceptível o comportamento para a matriz 4096x4096, em que a performance com 12 threads foi significativamente inferior à de 10 e 8 threads, indicando que o custo da contenção de recursos e o overhead superaram os benefícios de adicionar mais threads além do número de núcleos físicos.

3.2.2 Escalabilidade e Speedup

Com gráfico de linhas na Figura 2 é possível avaliar a escalabilidade da solução, mostrando como o speedup se comporta com o aumento do número de threads. A linha tracejada preta representa o "Speedup Ideal", onde o ganho de desempenho seria perfeitamente linear com o número de núcleos.



Fonte: Autoria própria.

Figura 2 – Relação entre Speedup e número de threads para diferentes tamanhos de matriz.

O gráfico mostra como o algoritmo se comporta com diferentes números de threads e tamanhos de matriz. A escalabilidade é boa até cerca de 8 a 10 threads, quando a maioria das linhas atinge o seu máximo de speedup. Depois disso, o comportamento muda. Por exemplo, as linhas vermelha (4096x4096) e verde (2048x2048) apresentam uma queda de desempenho ao passar de 10 para 12 threads. Isso acontece porque o processador tem 6 núcleos físicos e 12 threads, e para tarefas pesadas, as threads lógicas acabam competindo pelos mesmos recursos, gerando sobrecarga que pode anular os ganhos.

O efeito de speedup superlinear ainda aparece na linha laranja (1024x1024), que chega a mais de 13x com 12 threads. Isso indica que, para esse tamanho de problema, a divisão das tarefas melhorou bastante o uso da memória cache. Já os problemas pequenos, como a linha azul (512x512), mostram apenas pequenos ganhos e não apresentam quedas fortes, indicando que adicionar threads extras não ajuda muito, mas também não prejudica tanto o desempenho.

4 Discussão

Neste capítulo, os resultados quantitativos apresentados anteriormente são interpretados de forma qualitativa. A discussão aprofunda a comparação entre as abordagens sequencial e paralela, analisa o impacto da variação no número de threads e contextualiza os gargalos e limitações encontradas, finalizando com sugestões para trabalhos futuros.

4.1 Comparação entre as Versões Sequencial e Paralela

A comparação entre as versões sequencial e paralela mostrou a vantagem do paralelismo. Enquanto a versão sequencial fica limitada a um único núcleo, a paralela distribui a carga de trabalho pelos vários núcleos, reduzindo o tempo de execução de dezenas de segundos para apenas alguns. O caso mais marcante foi o da matriz 2048x2048, que apresentou um speedup superlinear: ao dividir o problema, cada thread passou a lidar com menos dados, que puderam ser armazenados inteiramente na memória cache, muito mais rápida que a RAM. Assim, o desempenho não veio só da execução simultânea, mas também de um uso mais eficiente da hierarquia de memória.

Por outro lado, os testes também mostraram que o aumento no número de threads não garante ganhos ilimitados. Até 8 threads, houve ganhos consistentes, principalmente nas matrizes maiores, evidenciando a boa escalabilidade do algoritmo. Porém, em certo momento, o desempenho começou a cair. O pico foi atingido com cerca de 10 threads, mas, ao chegar a 12, houve uma queda acentuada, já que as threads lógicas (SMT/Hyper-Threading) passaram a disputar os mesmos recursos de cada núcleo físico. Isso gerou contenção e overhead, anulando os benefícios da paralelização. Esses resultados deixam claro que o paralelismo pode trazer ganhos enormes, mas o número de threads precisa ser ajustado com cuidado, levando em conta tanto o hardware quanto o tipo de tarefa.

4.2 Gargalos e Limitações Encontradas

Com base na análise, foi possível perceber os principais gargalos e limitações que atrapalharam o desempenho. O primeiro deles foi a contenção de recursos causada pelo SMT em problemas grandes. Nos testes, quando se usaram 12 threads, o desempenho caiu porque o processador começou a usar de forma intensa as threads lógicas do SMT. Isso mostrou que, na prática, existe um limite no número de threads realmente úteis, e que esse limite é menor do que o total de threads lógicas disponíveis.

Outro ponto importante foi a largura de banda da memória. Apesar do efeito

positivo do cache, que até gerou o speedup superlinear, a velocidade com que os dados são transferidos da memória RAM para os núcleos do processador continua sendo um gargalo. Isso acontece porque, conforme mais núcleos passam a trabalhar ao mesmo tempo, cresce também a demanda sobre o barramento de memória, e ele pode se tornar o principal fator que limita o desempenho.

Por fim, o impacto do overhead (ou custo) da paralelização ficou muito claro. Isso foi especialmente visível na matriz menor, de 512x512, onde o ganho de desempenho foi bem mais modesto. Para problemas pequenos como este, o trabalho de criar, gerenciar e sincronizar as threads acaba sendo grande demais em comparação com o benefício de dividir a tarefa.

5 Conclusão

Este projeto teve como objetivo principal a implementação, otimização e análise comparativa de uma rotina de multiplicação de matrizes (DGEMM) em versões sequencial e paralela, utilizando a API OpenMP. Os objetivos foram alcançados, e a análise dos dados coletados permitiu extrair conclusões aprofundadas sobre a eficácia e os limites da computação paralela para problemas de alta demanda computacional.

As considerações sobre o desempenho obtido são em sua maioria positivas, mas revelam certa complexidade. A versão paralela demonstrou superioridade em relação à versão sequencial, alcançando um speedup máximo de **11x** para a maior matriz testada (4096x4096) com 10 threads, um resultado que supera em muito o número de núcleos físicos do processador. No entanto, a análise também expôs os limites práticos da paralelização neste hardware específico, com uma degradação de desempenho ao utilizar 12 threads para os problemas mais intensivos. Isso evidencia o impacto negativo da contenção de recursos ao empregar threads lógicas para esta carga de trabalho. O fenômeno do speedup superlinear, que atingiu um pico de mais de **13x** para a matriz de 1024x1024, permaneceu como um resultado de destaque, confirmando que a paralelização bem-sucedida pode otimizar drasticamente o uso da hierarquia de memória.

A execução do projeto permitiu consolidar tanto a prática quanto a teoria sobre paralelização. O principal aprendizado foi perceber a diferença entre aproveitar bem os núcleos físicos e os limites de desempenho ao usar threads lógicas. Quando o número de threads (12) ultrapassou o de núcleos físicos, o desempenho caiu, já que as threads passaram a disputar os mesmos recursos dentro de cada núcleo. Isso mostrou, na prática, que o SMT/Hyper-Threading pode atrapalhar em tarefas muito pesadas e que o overhead da paralelização não aparece apenas em problemas pequenos, mas também quando há competição de recursos. Assim, ficou claro que usar paralelismo de forma eficiente depende de analisar com cuidado a relação entre algoritmo, número de threads e arquitetura do hardware.

Referências

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: *AFIPS Conference Proceedings*. Atlantic City, NJ: AFIPS Press, 1967. v. 30, p. 483–485. Citado na página 9.

CHAPMAN, B. et al. *Using OpenMP: Portable Shared Memory Parallel Programming*. Cambridge, MA: MIT Press, 2021. Citado na página 9.

CORMEN, T. H. et al. *Introduction to Algorithms*. 3rd. ed. Cambridge, MA: MIT Press, 2009. Citado na página 9.

GRAMA, A. et al. *Introduction to Parallel Computing*. 2nd. ed. Harlow: Addison-Wesley, 2003. Citado na página 9.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. 6th. ed. Cambridge, MA: Morgan Kaufmann, 2019. Citado na página 9.