Universidade Estadual de Santa Cruz

Brenno S. Florêncio e Mateus Soares.

DGEMM Sequencial e Paralela com OpenMP

Trabalho apresentado à disciplina DEC107 – Processamento Paralelo, do curso de Ciência da Computação da Universidade Estadual de Santa Cruz, como requisito parcial de avaliação, sob orientação do professor Dr. Esbel Tomás Valero Orellana

Ilhéus - BA

28 de setembro de 2025

1. Introdução

A multiplicação de matrizes é uma das operações mais fundamentais da computação científica, utilizada em áreas como processamento de imagens, aprendizado de máquina, simulações numéricas e modelagem de sistemas físicos. Dentre os algoritmos disponíveis, a rotina **DGEMM** (Double-precision General Matrix Multiply) ocupa posição de destaque por operar com elementos em ponto flutuante de dupla precisão, garantindo maior acurácia em cálculos de alta complexidade.

O presente projeto tem como objetivo implementar e avaliar duas versões da rotina DGEMM: uma versão **sequencial**, sem uso de paralelismo, e uma versão **paralela**, utilizando diretivas do **OpenMP**. A implementação sequencial estabelece a base de comparação para o desempenho obtido com a versão paralela, enquanto a paralelização busca explorar arquiteturas de memória compartilhada, comuns em processadores multicore.

O OpenMP, por meio de diretivas como #pragma omp parallel e #pragma omp for, permite distribuir o trabalho entre múltiplos threads, reduzindo o tempo de execução da aplicação. A análise de desempenho realizada neste projeto considera métricas clássicas como **tempo de execução**, **speedup** e **eficiência**, que possibilitam avaliar o impacto do paralelismo no processamento de diferentes tamanhos de matrizes.

A importância desta análise reside não apenas na comparação entre as versões sequencial e paralela, mas também na compreensão dos **gargalos de desempenho** e da **escalabilidade** do algoritmo. Tais fatores são essenciais para a formação em Processamento Paralelo, visto que oferecem ao aluno a oportunidade de vivenciar na prática os conceitos teóricos estudados em sala de aula.

2. Metodologia

A implementação da rotina **DGEMM** foi desenvolvida em linguagem C com suporte ao **OpenMP** para paralelização. O objetivo metodológico foi manter a mesma lógica algorítmica nas versões sequencial e paralela, introduzindo otimizações de

localidade de memória e micro-otimizações simples, para então isolar o impacto do paralelismo na redução do tempo de execução.

Os experimentos foram executados em um computador pessoal equipado com processador **Intel® Core™ i5-10500H**, com **6 núcleos físicos e 12 threads**, frequência base de **2,50 GHz** e cache L1d: 192 KiB (6 instances) = 32Kb, L1i: 192 KiB (6 instances) = 32kb, L2: 1.5 MiB (6 instances) L3: 12 MiB (1 instance). O sistema contava com **8 GB de memória RAM** e arquitetura **x86_64**, suportando instruções vetoriais avançadas como **AVX2** e **FMA**, que favorecem operações de ponto flutuante.

O sistema operacional utilizado foi o **Debian GNU/Linux 12 (Bookworm)**, em ambiente de 64 bits. O compilador adotado para as implementações foi o **GCC versão 12.2.0**, com as opções de otimização -Wall -O3 -fopenmp -march=native -mfma. Essas configurações exploram tanto o paralelismo em múltiplos núcleos quanto a vetorização em nível de instrução, permitindo melhor aproveitamento do hardware disponível.

Representação dos dados. Todas as matrizes foram representadas em **arranjos unidimensionais** de double no formato row-major. O acesso a um elemento na linha i e coluna j é feito por i * n + j. O código emprega o qualificador restrict nos ponteiros para informar ao compilador a inexistência de aliasing, permitindo melhor vetorização e reordenação de acessos pela ferramenta de otimização.

Otimizações de memória: transposição e tiling. Para reduzir falhas de cache durante a multiplicação, a matriz **B** é transposta para um buffer bT antes do cálculo, tanto na versão sequencial quanto na paralela. Isso transforma o acesso originalmente não contíguo a B em acesso contíguo a B^t, melhorando o throughput de memória. Além disso, foi aplicado **block tiling** com tamanho de bloco fixo nos três eixos (i, j, k). A blocagem limita a working set por bloco, aumenta o reuso em cache L1/L2 e reduz cache misses. A transposição também é realizada com tiling, evitando percursos longos em memória. O blockSize = 64 foi escolhido para caber no L1 (mencionado no incio da metodologia), 64×64×8 bytes (tamanho do tipo double em c =32 KiB, exatamente o tamanho típico do L1d cache por core, caso

fosse escolhido o blockSize = 128 ou maior: 128×128×8=128 kb, já estoura o L1, indo para L2 o que aumentaria a latência e reduziria levemente a eficiência,

Register Block de 2×2. Dentro de cada bloco, foi utilizado um pequeno unroll de 2×2 (acúmulo de c00, c01, c10, c11) para reduzir a pressão de leitura/escrita na matriz C e aumentar o reuso de elementos de A e B^t nos registradores. Essa técnica diminui o número de acessos à memória principal e permite ao compilador explorar melhor a vetorização.

Versão sequencial. A rotina dgemmSeq realiza: (i) transposição de B em bT com tiling; (ii) três laços encaixados blocados (ii, jj, kk) e laços internos com unroll 2×2, acumulando produtos parciais em registradores; (iii) escrita final em C com o fator alpha. As matrizes C são alocadas com calloc, iniciando em zero, de modo que o parâmetro beta (definido como 0.0) permanece apenas por compatibilidade conceitual com a interface DGEMM, sem efeito prático no acúmulo.

Versão paralela (OpenMP). A paralelização preserva a mesma estrutura algorítmica e introduz diretivas OpenMP em dois pontos:

(i) **Transposição paralela**: transposePar utiliza #pragma omp parallel for collapse(2) sobre os blocos (ii, jj) para distribuir o trabalho entre threads .

(ii) **Multiplicação paralela**: dgemmPar aplica #pragma omp parallel for collapse(2) private(i,j,k,ii,jj,kk) num_threads(p) sobre os laços de blocos (ii, jj), gerando granularidade suficiente para escalar em múltiplas threads. As variáveis de laço são privadas para evitar condições de corrida. Como cada iteração escreve posições distintas de C, não há necessidade de seções críticas; a sincronização ocorre apenas nas barreiras implícitas ao final dos loops paralelos. O schedule padrão do OpenMP (static) foi mantido, por ser adequado a iterações uniformes.

Geração de dados e corretude: As matrizes de entrada são preenchidas com valores pseudoaleatórios uniformes rand()/RAND_MAX. Para validar a corretude, o programa calcula a soma absoluta das diferenças elemento a elemento entre a saída sequencial e cada saída paralela; em todos os experimentos essa diferença foi zero, confirmando a equivalência numérica das versões.

Procedimento de medição: O tempo de execução foi medido com omp_get_wtime() envolvendo exclusivamente a chamada de cada rotina (dgemmSeq ou dgemmPar). Importante destacar que o tempo da transposição está incluído em cada medição, refletindo o custo real da estratégia adotada. Para cada tamanho de matriz, o programa executa a versão sequencial e, em seguida, as versões paralelas com diferentes quantidades de threads, registrando os tempos imediatamente após cada execução.

Conjuntos experimentais: Atendendo ao enunciado, foram testados tamanhos de matrizes quadradas **512, 1024, 2048 e 4096**. Para a versão paralela, foram avaliadas configurações com **2, 4, 8 e 12 threads**, explorando os recursos do hardware disponível. As métricas analisadas incluem tempo de execução, speedup (razão entre o tempo sequencial e o paralelo) e eficiência (speedup dividido pelo número de threads).

3. Resultados

Nesta seção são apresentados os resultados obtidos com as execuções da rotina DGEMM nas versões sequencial e paralela, utilizando diferentes tamanhos de matrizes e quantidades de threads. Os experimentos foram conduzidos de forma consistente, garantindo que os tempos de execução se referem apenas à multiplicação das matrizes, incluindo a etapa de transposição.

A **Tabela 1** resume os valores medidos para cada configuração, apresentando o tempo de execução, o speedup em relação à versão sequencial e a eficiência correspondente. Como esperado, a versão paralela apresentou desempenho superior em todos os cenários, com ganhos crescentes à medida que o número de threads aumentou.

Tamanho da matriz	Threads	Tempo (s)	Speedup	Eficiência	Observação
512	1	0.038883	1.000	1.000	Sequencial
512	2	0.021852	1.779	0.890	Paralelo
512	4	0.011004	3.534	0.883	Paralelo
512	8	0.010132	3.838	0.480	Paralelo
512	12	0.011202	3.471	0.289	Paralelo
1024	1	0.292792	1.000	1.000	Sequencial
1024	2	0.147527	1.985	0.992	Paralelo
1024	4	0.087228	3.357	0.839	Paralelo
1024	8	0.060021	4.878	0.610	Paralelo
1024	12	0.050556	5.791	0.483	Paralelo
2048	1	2.245876	1.000	1.000	Sequencial
2048	2	1.211061	1.854	0.927	Paralelo
2048	4	0.739238	3.038	0.760	Paralelo
2048	8	0.417283	5.382	0.673	Paralelo
2048	12	0.353428	6.355	0.530	Paralelo
4096	1	17.745193	1.000	1.000	Sequencial
4096	2	10.505735	1.689	0.845	Paralelo
4096	4	5.696381	3.115	0.779	Paralelo
4096	8	3.681408	4.820	0.603	Paralelo
4096	12	3.287597	5.398	0.450	Paralelo

A corretude das implementações foi validada por meio da comparação elemento a elemento entre os resultados da versão sequencial e das versões paralelas, sendo constatada diferença nula em todas as execuções. Esse resultado confirma que a paralelização preservou a exatidão do algoritmo.

Além da tabela, os resultados foram representados graficamente para facilitar a visualização das tendências:

Figura 1 - Tempo de execução vs Threads (escala logarítmica): Apresenta-se, na Figura 1, a relação entre o tempo de execução e o número de threads utilizadas para diferentes tamanhos de matrizes. Os resultados evidenciam que o aumento do grau de paralelismo reduz significativamente o tempo total de execução em todos os cenários.

Verifica-se que o impacto da paralelização é mais expressivo para matrizes de maior dimensão, como N=2048 e N=4096, nas quais a queda no tempo é bastante acentuada conforme se incrementa o número de threads. Para matrizes menores, como N=512 e N=1024, a redução também ocorre, mas com ganhos marginais à medida que se ultrapassa um certo número de threads, indicando que o custo de gerenciamento paralelo se torna comparável ao tempo gasto no cálculo.

Em particular, no caso de N=4096, o tempo de execução passa de aproximadamente 17 segundos na versão sequencial para pouco mais de 3 segundos com 12 threads, o que demonstra uma boa escalabilidade, embora não linear. Esses resultados confirmam que o uso do paralelismo em arquiteturas de memória compartilhada é especialmente vantajoso quando aplicado a problemas de maior porte, enquanto em matrizes menores o benefício tende a ser limitado pela sobrecarga de sincronização e coordenação entre as threads.

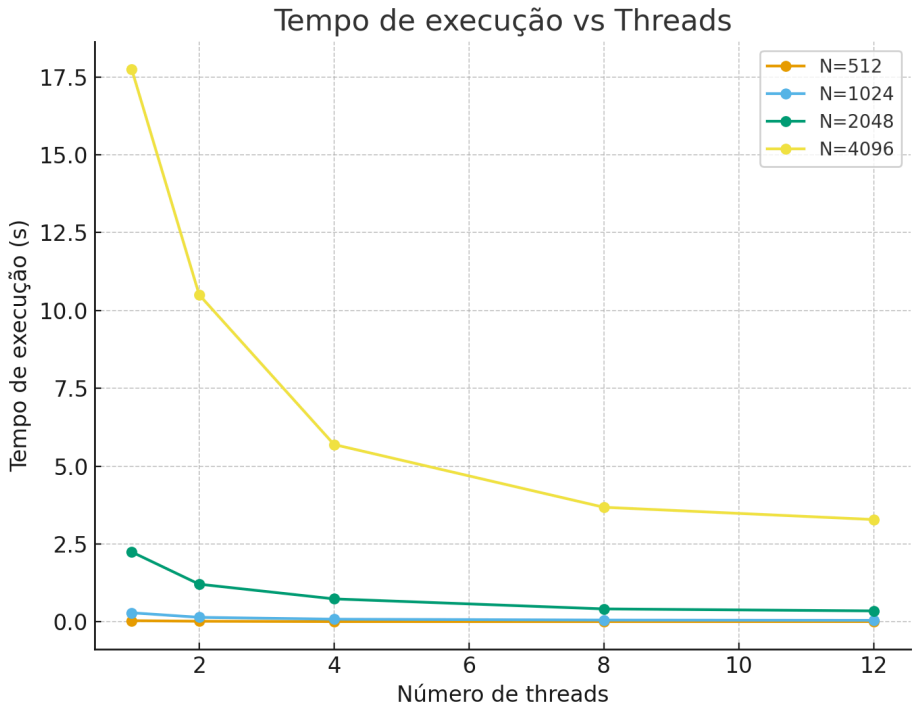


Figura 2 - Speedup vs Threads: Apresenta-se, na Figura 2, o comportamento do speedup em função do número de threads para diferentes tamanhos de matrizes. O speedup representa o ganho de desempenho da versão paralela em relação à versão sequencial.

Observa-se que, de modo geral, o speedup cresce conforme se aumentam as threads, mas o crescimento não é linear. Para matrizes de maior dimensão (N=2048 e N=4096), o ganho é mais consistente, aproximando-se da linha de speedup ideal nas primeiras configurações e mantendo crescimento progressivo até 12 threads. Isso evidencia que problemas maiores conseguem explorar melhor os recursos de paralelismo.

Para N=1024, o comportamento também é positivo, embora com ganhos ligeiramente inferiores aos observados em N=2048. Já no caso de N=512, nota-se uma saturação precoce: o speedup cresce até cerca de 8 threads, mas depois sofre queda, o que sugere que a sobrecarga de coordenação entre as threads se torna mais significativa que os benefícios obtidos pelo paralelismo.

De forma geral, os resultados indicam que o paralelismo é mais vantajoso em matrizes de maior porte, nas quais a quantidade de operações compensa a

sobrecarga, enquanto em matrizes pequenas o ganho se estabiliza rapidamente e pode até regredir.

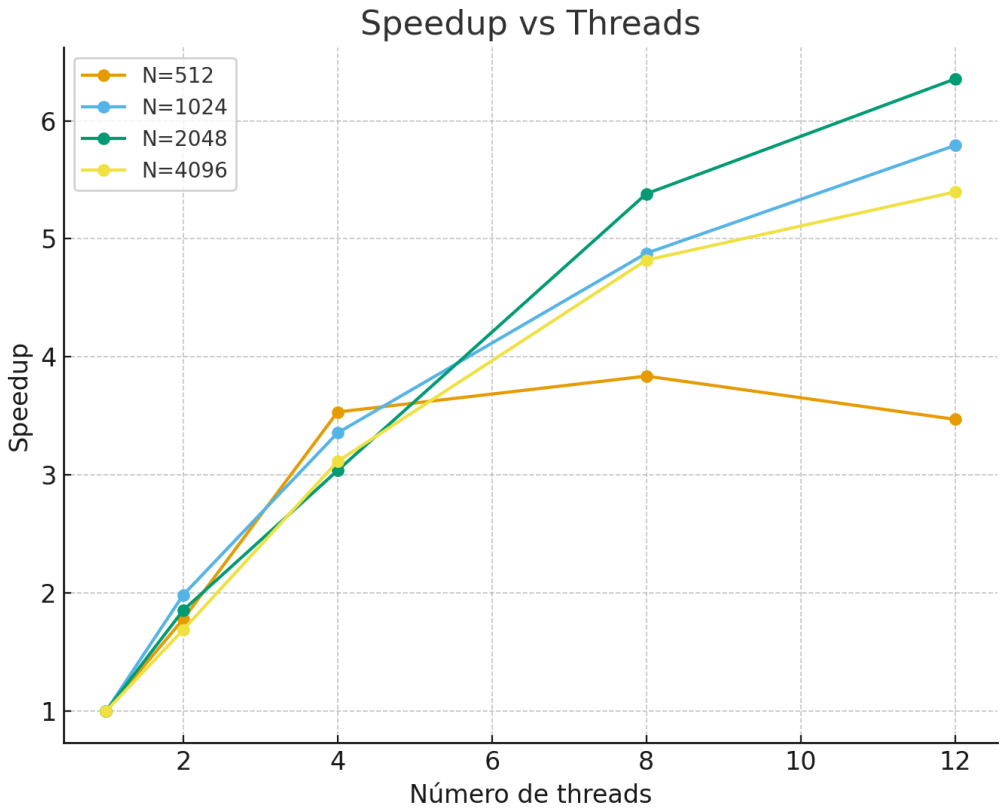


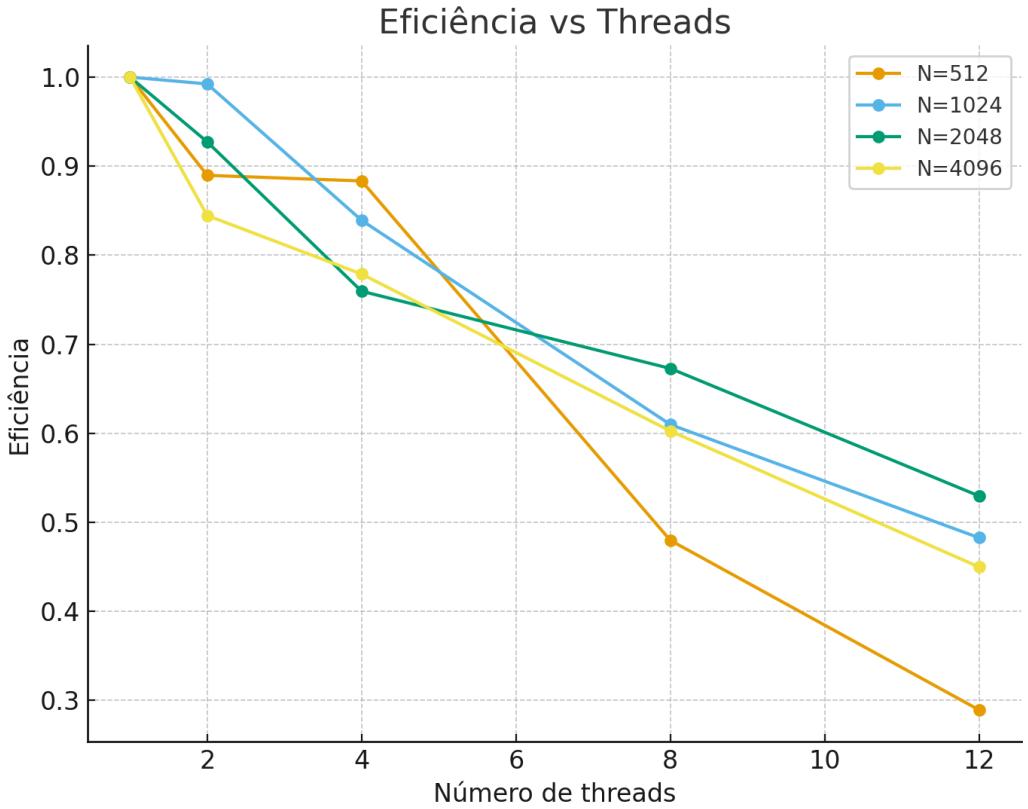
Figura 3 - Eficiência vs Threads: Apresenta-se, na Figura 3, a variação da eficiência em função do número de threads para diferentes tamanhos de matrizes. A eficiência mede o quanto cada thread é efetivamente aproveitada, sendo obtida pela razão entre o speedup e a quantidade de threads.

Observa-se que, em todos os casos, a eficiência diminui progressivamente à medida que o número de threads aumenta. Esse comportamento é esperado em arquiteturas de memória compartilhada, pois o custo de sincronização, a contenção por acesso à memória e o gerenciamento do paralelismo reduzem o aproveitamento ideal.

As matrizes de maior dimensão (N=2048 e N=4096) apresentam melhor manutenção da eficiência em altos números de threads, permanecendo acima de 50% mesmo com 12 threads. Isso ocorre porque o volume de operações compensa

parte da sobrecarga. Já para matrizes menores (N=512), a queda é mais acentuada: a eficiência cai para valores próximos de 30% em 12 threads, indicando que o custo de paralelizar supera os ganhos de processamento.

Em síntese, o gráfico confirma que a eficiência é inversamente proporcional ao número de threads, sendo mais bem sustentada em problemas de maior porte, nos quais há maior paralelismo intrínseco a ser explorado.



4. Discussão

A comparação entre as versões sequencial e paralela demonstra ganhos claros de desempenho com a utilização do OpenMP. A versão sequencial serviu como linha de base, enquanto a versão paralela reduziu significativamente o tempo de execução em todos os tamanhos de matrizes testados. Mesmo em matrizes menores, em que o custo computacional é relativamente baixo, a paralelização já trouxe benefícios. À medida que o tamanho das matrizes aumenta, o impacto do paralelismo se torna ainda mais expressivo, confirmando que problemas de maior porte se beneficiam mais da execução concorrente.

O aumento no número de threads também apresentou efeito positivo, embora não linear. Até um certo ponto, a escalabilidade se manteve próxima do ideal, mas a partir de um número maior de threads os ganhos adicionais se tornaram menores. Esse comportamento é esperado em arquiteturas de memória compartilhada, pois o acesso concorrente à memória e a necessidade de sincronização entre as threads passam a ser fatores limitantes.

Alguns gargalos e limitações foram identificados durante os testes. Entre eles, destacam-se a sobrecarga introduzida pelo gerenciamento de threads, o consumo elevado de largura de banda de memória e a saturação da hierarquia de cache em matrizes muito grandes. Esses fatores reduzem a eficiência da paralelização, impedindo que os ganhos cresçam proporcionalmente ao número de threads.

Para mitigar essas limitações, algumas estratégias podem ser consideradas. Ajustar o tamanho dos blocos utilizados na multiplicação pode melhorar o aproveitamento do cache. Testar diferentes políticas de escalonamento do OpenMP, como dynamic ou guided, pode favorecer o balanceamento de carga em cenários menos uniformes. Além disso, em trabalhos futuros, seria possível explorar técnicas mais avançadas, como aumento do resgister blocking para 4x4, ou MXN(dependendo muito dos Hardware), Reempacotamento de dados (Packing), Vetorização, ou aceleração em GPU, ampliando a escalabilidade do algoritmo.

Outra estratégia utilizada para amenizar parte desses problemas foi o emprego da técnica de **blocagem (*tiling*)**, definida no código pelo parâmetro blockSize. Essa abordagem permite que as operações sejam realizadas em blocos menores de dados, favorecendo a localidade espacial e temporal e, consequentemente, um melhor aproveitamento dos níveis de cache (L1, L2 e L3). Apesar disso, a escolha do tamanho do bloco envolve um equilíbrio delicado: blocos muito grandes podem não caber inteiramente no cache, aumentando a latência, enquanto blocos muito pequenos elevam a sobrecarga de gerenciamento. Portanto, a parametrização adequada do blockSize é essencial para maximizar o desempenho, sobretudo em matrizes de maior dimensão.

5. Conclusão

O desenvolvimento deste projeto possibilitou aplicar de forma prática os conceitos de processamento paralelo em arquiteturas de memória compartilhada, consolidando os conteúdos estudados em sala. A implementação da rotina DGEMM em versões sequencial e paralela permitiu compreender tanto a lógica básica da multiplicação de matrizes quanto os desafios associados à sua otimização. O uso de técnicas como transposição de matrizes e blocagem mostrou-se essencial para melhorar a localidade de memória, reforçando a importância das estratégias de acesso eficiente em aplicações de alto desempenho.

Do ponto de vista do paralelismo, o projeto evidenciou os benefícios diretos da utilização do OpenMP na redução do tempo de execução, bem como os limites práticos impostos pelo hardware. Ficou claro que, embora a paralelização traga ganhos expressivos, esses ganhos não crescem indefinidamente com o aumento do número de threads, devido a fatores como sobrecarga de gerenciamento e competição por recursos de memória.

Em termos de aprendizado, a experiência destacou a relevância de avaliar criticamente métricas de desempenho: tempo de execução, speedup, eficiência e de compreender o impacto da arquitetura do sistema na execução paralela. Como consideração final, pode-se afirmar que a paralelização é altamente vantajosa para problemas de maior porte, mas exige um equilíbrio cuidadoso entre algoritmos, recursos disponíveis e técnicas de otimização para que o desempenho obtido seja próximo do ideal.

6. Referências

1. CONEGLIANO, Andrew; SPRINGER, Matthias. Double-precision General Matrix Multiply (DGEMM), Parallel Computation (CSE 260), Assignment 1. 2014. 9 f. Disponível em: https://m-sp.org/downloads/cse260_a1.pdf
2. OPENBLAS DEVELOPERS. OpenBLAS: An optimized BLAS library. Disponível em: https://www.openblas.net/. Acesso em: 26 set. 2025.
3. INTEL CORPORATION. Intel oneAPI Math Kernel Library (oneMKL) Developer Reference. 2025. Disponível em: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/. Acesso em: 26 set. 2025.
4. GOTO, Kazushige; VAN DE GEIJN, Robert. Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software (TOMS), v. 34, n. 3, p. 1–25, 2008. DOI: https://doi.org/10.1145/1356052.1356053.
5. OPENMP ARCHITECTURE REVIEW BOARD. OpenMP Application Programming Interface. Version 5.2, 2021. Disponível em: https://www.openmp.org/specifications/. Acesso em: 26 set. 2025.
6. BLANCHARD, Pierre; GU, Zeyu; HIGHAM, Nicholas J.; RAINBOW, Michael A. Parallelizing matrix multiplication with OpenMP. University of Manchester, 2020. Disponível em: https://arxiv.org/abs/2008.06817. Acesso em: 26 set. 2025.
7. WIKIPEDIA. General Matrix Multiply (GEMM). 2025. Disponível em: https://en.wikipedia.org/wiki/General_Matrix_Multiply. Acesso em: 26 set. 2025.