SIMPAR
Simpósio de Pesquisa, Extensão e Inovação do Paraná
Realização
IN2 Instituto Integrado de Ciência e Tecnologia
Centro Universitário Integrado
Núcleo de Empreendedorismo, Pesquisa e Extensão Integrado
Apoio
Fundação Araucária
Apoio ao Desenvolvimento Científico e Tecnológico do Paraná

Análise de Desempenho de Técnicas Paralelas e Vetorizadas na Multiplicação de Matrizes

Hiago Gimenez Vieira, Curso Técnico Integrado em Informática para Internet, UTFPR, Brasil

João Fabrício Filho, Professor, UTFPR, Brasil, joaof@utfpr.edu.br

Este trabalho investiga o desempenho de diferentes técnicas na multiplicação de matrizes, com foco em computação paralela e vetorização. Foram comparados dois algoritmos, o método trivial e o algoritmo de *Strassen*, sendo avaliados em conjunto com abordagens como blocagem (*blocking*), vetorização *SIMD(AVX)*, e bibliotecas potencializadas (*BLAS* e *EIGEN*). Os testes foram realizados com diferentes tamanhos de matrizes, número de *threads*, e duas *APIs* de paralelismo, *OpenMP* e *Pthreads*. Os resultados mostram que o uso de blocagem adequada, vetorização e bibliotecas pode superar o desempenho de implementações simples. O método trivial, quando aplicado com técnicas de paralelismo, supera o algoritmo de *Strassen* em até 2 vezes. O *OpenMP* apresenta maior praticidade, enquanto que *Pthreads* demonstra maior estabilidade em cenários potencializados. As bibliotecas *BLAS* e *EIGEN* se destacam pelos maiores ganhos de desempenho, com *speedup* de até 900x. Este estudo contribui para a escolha criteriosa de técnicas de paralelização e técnicas em aplicações de multiplicação de matrizes em arquiteturas *multicore*.

Palavras-chave: Computação paralela; Multiplicação de matrizes; *Speedup*; *OpenMP*; *Pthreads*;

This work investigates the performance of different techniques in matrix multiplication, focusing on parallel computing and vectorization. Two algorithms were compared, the trivial method and the Strassen algorithm, and evaluated together with approaches such as blocking, SIMD (AVX) vectorization, and optimized libraries (BLAS and EIGEN). The experiments were conducted using different matrix sizes, numbers of threads, and two parallelism APIs: OpenMP and Pthreads. The results show that proper use of blocking, vectorization, and libraries can outperform simple implementations. The trivial method, when applied with parallel programming methods, outperforms the Strassen algorithm by up to 2×. OpenMP proved to be more practical, while Pthreads shows greater stability in optimized scenarios. The BLAS and EIGEN libraries stood out by delivering the highest performance gains, achieving speedups of up to 900×. This study contributes to the careful selection of parallelization strategies and techniques for matrix multiplication applications on multicore architectures.

Keywords: *Parallel computing; Matrix multiplication; Speedup; OpenMP; Pthreads;*

INTRODUÇÃO

Há aplicações que necessitam de poder de processamento com alto desempenho para serem executadas em tempo hábil, como, por exemplo, a previsão do tempo e simulações físicas, as quais levam dias ou até meses quando executadas sequencialmente (1).

A modelagem de problemas utilizando matrizes é de extrema importância para a ciência da computação. Áreas como computação gráfica, grafos e aprendizado de máquina utilizam matrizes com alta frequência para solucionar seus respectivos problemas (2).

Existem diferentes métodos para realizar a multiplicação de matrizes, variando em complexidade e desempenho. Entre eles, destacam-se o algoritmo trivial, por sua simplicidade e fácil implementação, e o algoritmo de *Strassen (3)*, que utiliza uma abordagem recursiva para reduzir o número de operações necessárias. Esses métodos são amplamente estudados para avaliar o impacto de potencialização e técnicas de paralelismo.

Com base em resultados anteriores (13), este trabalho, analisa implementações diferentes de dois algoritmos de multiplicação de matrizes: o método trivial e o algoritmo de *Strassen.* Devido à sua estrutura recursiva, o algoritmo de *Strassen* se divide em 7 multiplicações independentes, assim o paralelismo utilizando *Pthreads* (4) foi limitado a 7 *threads*. Isso ocorre porque, a cada nível de recursão, o algoritmo de *Strassen* gera até sete multiplicações de matrizes que podem ser executadas em paralelo de forma independente (5, 6).

Para explorar o paralelismo, foram utilizadas duas *APIs*, *OpenMP* (7), que oferece uma abordagem de paralelização por diretivas de compilador, e *Pthreads*, que permite a criação manual de *threads*. Cada técnica foi avaliada em matrizes de tamanhos como *512x512*, *1024x1024*, *2048x2048* e *4096x4096* e foram executadas com 2,4, 8 e 16 *threads*, exceto no caso do *Strassen* com *Pthreads*, no qual cada teste foi executado 10 vezes, com o objetivo de medir o tempo de execução médio, o desvio padrão e o ganho de *speedup* (ganho de velocidade obtido de um algoritmo, calculado com tempo sequencial dividido pelo tempo com técnica).

Além do paralelismo, o trabalho investigou o impacto de estratégias de blocagem (*blocking*) (8), com tamanhos de bloco 64 e 128, visando maximizar a localidade de *cache*. De vetorização, foram exploradas instruções *SIMD(AVX)* (9) e bibliotecas de álgebra linear potencializadas, como *BLAS* (10) e *EIGEN* (11), cada uma com suas configurações de compilação, *-O2* para *BLAS(-lopenblas)* e *EIGEN*, *-mavx* para *SIMD*, e *-O* para as versões somente paralelas.

Em termos de desempenho, os resultados mostram que, para uma matriz *2048x2048*, o método trivial sequencial teve tempo médio de 63,30 segundos, enquanto a versão *OpenMP + Blocking(64) + SIMD(AVX)* com 16 *threads* alcançou 1,11 segundos, um *speedup* de 56,76x. Para o mesmo tamanho de matriz, a combinação com *BLAS* supera os demais, com tempos abaixo de 0,06 segundos e *speedups* acima de 900x, evidenciando a eficiência das bibliotecas potencializadas. Já no *Strassen*, observa-se que, com blocagem e vetorização quaisquer, mesmo com *Pthreads* limitados, se reduzem os tempos de execução, com *speedups* acima de 25x em matrizes *1024x1024*.

Diante disso, este trabalho mostra a comparação entre algoritmos, *APIs* de paralelismo, técnicas de blocagem, vetorização e bibliotecas potencializadas, destacando qual abordagem apresenta os melhores resultados em cada cenário. Esses dados são essenciais para orientar decisões futuras na escolha de estratégias de otimização para aplicações em ambientes *multicore*.

MÉTODO

O algoritmo trivial de multiplicação de matrizes consiste em três laços de repetição aninhados, nos quais cada elemento da matriz resultante é obtido pela soma dos produtos correspondentes entre linhas da primeira matriz e colunas da segunda. Esse método possui complexidade computacional de $O(n\ 3)$ e serve como base para a maioria das otimizações em multiplicação de matrizes, devido à sua simplicidade e fácil implementação. O algoritmo de *Strassen*, proposto em 1969, é uma técnica recursiva que reduz a complexidade da multiplicação de matrizes de $O(n\ 3)$ para aproximadamente $O(n\ 2{,}81)$. Ele substitui parte das multiplicações por somas e subtrações, sendo vantajoso para matrizes grandes, teoricamente (3).

As implementações dos algoritmos trivial e *Strassen* se deram por meio das ferramentas abaixo:

- *POSIX Threads (Pthreads)* é uma *API* padronizada para programação concorrente em *C/C++*, definida pela *IEEE* (4). Ela permite a criação e gerenciamento explícito de múltiplas *threads* em ambientes com memória compartilhada, oferecendo controle fino sobre sincronização, escalonamento e divisão de tarefas entre núcleos de *CPU*.
- *Open Multi-Processing (OpenMP)* é uma *API* de paralelismo baseada em diretivas, amplamente utilizada para paralelizar laços e regiões de código em linguagens como *C*, *C++* e *Fortran*. Segundo o *OpenMP Architecture Review Board* (7), ela fornece uma interface de alto nível para programação paralela em arquiteturas de memória compartilhada, com suporte a diversos recursos modernos como paralelismo em vetorização.
- *BLOCKING* Consiste em reorganizar a multiplicação de matrizes de forma que os dados sejam processados em blocos menores. Isso melhora a localidade temporal e espacial dos dados, potencializando o uso de *caches* e reduzindo a latência de memória. *Goto e Van De Geijn* (8) mostraram que essa abordagem é essencial para alcançar alto desempenho em arquiteturas modernas.
- *Single Instruction Multiple Data (SIMD)* é um modelo de paralelismo que permite a execução de uma única instrução sobre múltiplos dados simultaneamente. A *Intel* (9) descreve instruções *SIMD* modernas, como *AVX(Advanced Vector Extensions)*, que são fundamentais para acelerar operações matemáticas em vetores e matrizes.

- *Basic Linear Algebra Subprograms (BLAS)* é uma interface padronizada para operações básicas de álgebra linear, como produtos matriz-matriz, matriz-vetor e operações escalares. Conforme *Lawson et al.* (10), a padronização das rotinas *BLAS* permite a criação de implementações altamente eficientes, base para bibliotecas de alto desempenho.
- *Eigen* é uma biblioteca *C++* baseada em templates para computação linear, que oferece suporte a matrizes, vetores, decomposições e resolução de sistemas lineares. A versão atual *Guennebaud et al.* (11) utilizam técnicas como *expression* templates e vetorização com *SIMD* para alcançar alto desempenho sem comprometer a legibilidade.

Os experimentos foram conduzidos nesses parâmetros:

• Processador: *Intel Core i5-13400 (16 threads)*

• Memória: *16GB DDR4*

• Sistema Operacional: *Linux Ubuntu 20.04 LTS*

• Compilador:

– Somente Paralelização (implementações em *C*): *GCC -O (-lpthread ou - openmp)*

– *SIMD (AVX, em C): GCC -O -mavx*

– *BLAS (em C): GCC -O2 -lopenblas*

– *EIGEN (em C++): G++ -O2*

Todas as implementações e *scripts* estão disponíveis em repositório público *(https://github.com/gimenezhiago/IniciacaoCientifica)*.

RESULTADOS E DISCUSSÃO

Os resultados obtidos nos experimentos mostram como diferentes abordagens influenciam o desempenho da multiplicação de matrizes. A comparação entre os dois algoritmos, o método trivial e o algoritmo de *Strassen*, mostra que o trivial, apesar de sua maior complexidade assintótica, apresenta tempos de execução menores na maioria dos cenários quando associado a técnicas como blocagem e uso de bibliotecas como *BLAS* e *EIGEN*. Por outro lado, o algoritmo de *Strassen*, mesmo com sua vantagem teórica de complexidade, tem limitações de restrição de paralelismo o que compromete sua escalabilidade em matrizes maiores. Ainda assim, *Strassen* mostrou desempenho competitivo em matrizes menores como *512x512*, sobretudo quando associado a blocagem e vetorização, capaz de atingir tempos como 0,0399 segundos em *Pthreads + Blocking(128) + SIMD(AVX)*, ficando próximo às melhores versões do método ingênuo.

Ao comparar as duas *APIs* de paralelização, *OpenMP* e *Pthreads*, observa-se que ambas apresentam resultados semelhantes em configurações básicas, com pequenas variações nos tempos de execução. No entanto,

cenários mais eficientes, com uso de blocagem e vetorização, a implementação com *Pthreads* apresentou desempenho superior em vários casos como demonstra a figura 1 e 2. Por exemplo, para a multiplicação trivial com matriz de *2048x2048* usando *Blocking(128)* e *SIMD(AVX)* com 16 *threads*, o tempo de execução com *Pthreads* foi de 1,1177 segundos, enquanto com *OpenMP* foi de 1,0927, valores próximos, mas com desvio padrão menor no caso de *Pthreads*, indicando maior estabilidade. Essa diferença de estabilidade foi ainda mais observável em testes com blocagem e bibliotecas *BLAS* e *EIGEN*, em que a versão com *Pthreads* frequentemente obteve melhores resultados, principalmente com 16 *threads*, como no caso da combinação com *BLAS*, em que a versão *Pthreads + Blocking(64) + BLAS* atingiu 0,0665 segundos, contra 0,0692 segundos da versão com *OpenMP*.

Em vetorizações, três abordagens foram exploradas: instruções *SIMD* (com *AVX*), e as bibliotecas *BLAS* e *EIGEN*. A vetorização com *AVX* apresentou ganhos consistentes, principalmente para matrizes de tamanho médio, como *1024x1024* e *2048x2048*, nas quais reduz os tempos de execução. Por exemplo, para *1024x1024* com *Pthreads + Blocking(128) + SIMD(AVX)* e 16 *threads*, o tempo foi de 0,1501 segundos, representando um *speedup* de 21x em relação ao sequencial. Porém, os maiores ganhos vieram das bibliotecas potencializadas. O uso do *BLAS* apresenta os melhores resultados em quase todos os casos. Em *2048x2048*, a versão *OpenMP + Blocking(64) + BLAS* com 16 *threads* atingiu apenas 0,0692 segundos, um *speedup* de mais de *900x*. Já com a biblioteca *EIGEN*, os ganhos foram igualmente expressivos, com desempenho aumentando conforme o número de *threads*, para o mesmo tamanho de matriz e configuração equivalente, foi obtido um tempo de 0,1276 segundos com *OpenMP*, e 0,1260 segundos com *Pthreads*, com *speedups* superiores a *490x* em ambos os casos. Mostrando que o uso de bibliotecas potencializadas supera qualquer abordagem baseada apenas em paralelismo ou blocagem, sendo especialmente eficiente em máquinas com múltiplos núcleos e suporte vetorial avançado.

Em relação ao tamanho dos blocos utilizados nas técnicas de blocagem, os testes revelam que blocos de tamanho 128 apresentam melhor desempenho em matrizes grandes, devido ao melhor aproveitamento da hierarquia de memória *cache*. Por exemplo, para *4096x4096*, a versão *OpenMP + Blocking(128) + SIMD* com 16 *threads* obteve tempo de 8,92 segundos, contra 11,30 segundos com blocos de tamanho 64, representando uma diferença significativa quando se considera o custo computacional total. Já em matrizes menores, o uso de blocos de tamanho 64 demonstrou resultados ligeiramente superiores, já que o *overhead* de alocação e movimentação de blocos grandes supera os benefícios de *cache*, mostrado por tempos de 0,0375 segundos com *OpenMP + Blocking(64) + SIMD(AVX)* com 16 *threads*, contra 0,0409 segundos na versão com blocos 128 na matriz *512x512*.

A variação no número de *threads* impactou diretamente no desempenho das execuções. O aumento do número de *threads* resultou em ganhos

consideráveis de *speedup*, sobretudo entre 2 e 8 *threads*. No entanto, entre 8 e 16 *threads*, observa-se um ganho pequeno em vários cenários, mostrando o efeito de saturação dos recursos da *CPU* e do barramento de memória. Por exemplo, para a matriz *1024x1024* com *OpenMP* + *Blocking(64)* + *SIMD(AVX)*, o tempo caiu de 0,2049 segundos com 8 *threads* para 0,1447 segundos com 16 *threads*, um ganho relativo pequeno em comparação com a queda entre 2 e 4 *threads* de 0,4943 segundos para 0,2571 segundos. No entanto, com *BLAS* ou *EIGEN*, os ganhos continuam crescendo de forma quase linear até 16 *threads*, como demonstrado Strassen pelos tempos obtidos com *BLAS*, em *2048x2048*, 0,2124 segundos (2 *threads*), 0,1199 segundos (4 *threads*), 0,0927 (8 *threads*) e 0,0692 segundos (16 *threads*). Indicando que tais bibliotecas potencializadas conseguem explorar melhor o paralelismo interno e a vetorização, e realizam um balanceamento mais eficiente entre os núcleos.

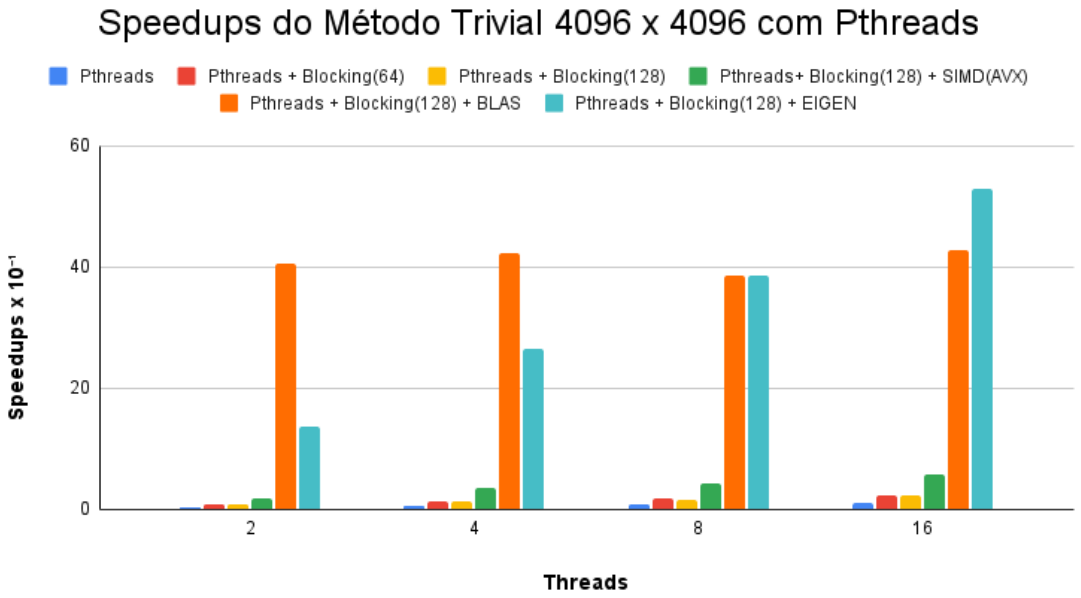

Figura 1 - *Speedups* do Método Trivial *4096x4096* com *Pthreads*. Comparação de diferentes técnicas: paralelização básica, blocagem (64/128), *SIMD(AVX)*, *BLAS* e *EIGEN* versus número de *threads*

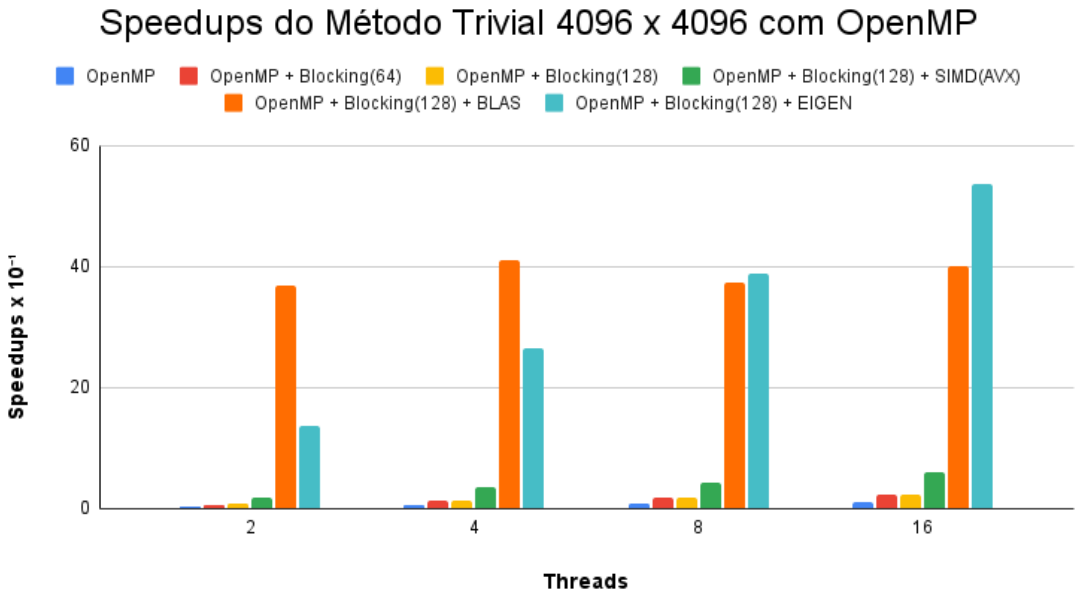

Figura 2 - *Speedups* do Método Trivial *4096x4096* com *OpenMP*. Mesmas técnicas da Figura 1, mostrando desempenho similar entre *APIs*, com *BLAS* e *EIGEN* superiores

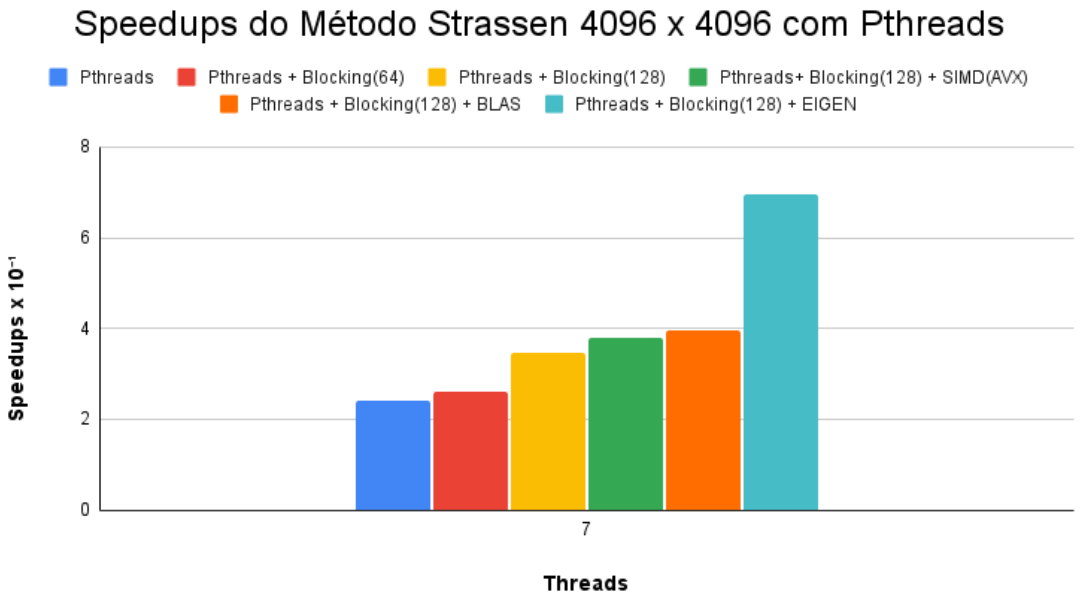

Figura 3 - *Speedups* do Método *Strassen 4096x4096* com *Pthreads*. Limitado pela recursão, máximo *140x* com *EIGEN*

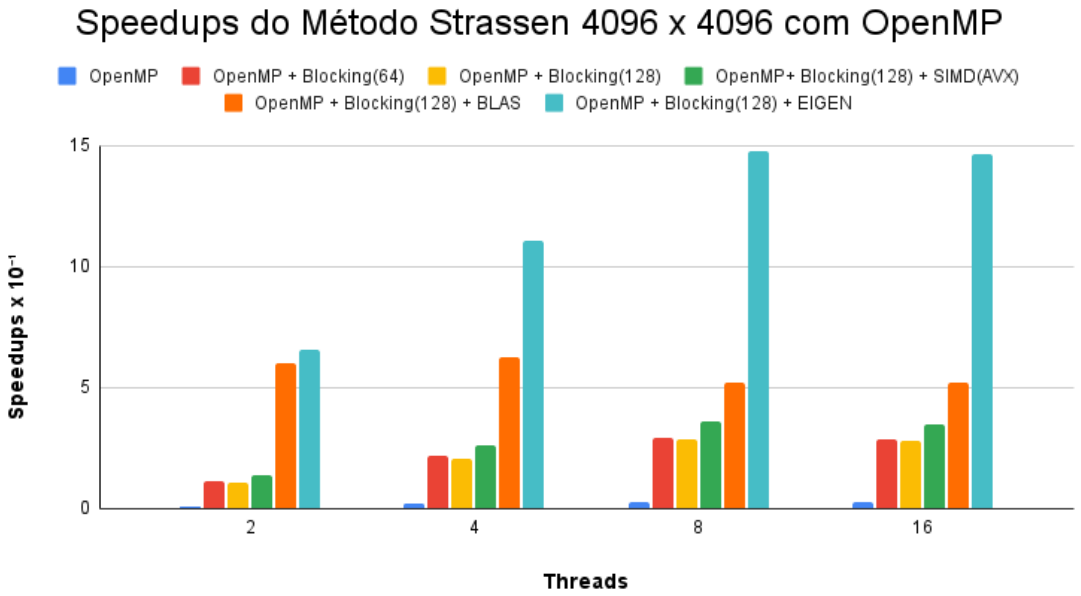

Figura 4 - *Speedups* do Método *Strassen 4096x4096* com *OpenMP*. Confirma limitações do *Strassen versus* método trivial

As Figuras 1, 2, 3 e 4 mostram que o método trivial, com técnicas como blocagem, vetorização e uso de bibliotecas *BLAS* e *EIGEN*, obteve os maiores *speedups*, superando *600x* com 16 *threads*. *EIGEN* destacou-se pela escalabilidade, enquanto *BLAS* apresentou ótimo desempenho até 8 *threads*.

Entre as *APIs*, *OpenMP* foi mais prático, mas *Pthreads* teve leve vantagem em estabilidade nos cenários potencializados. Já o algoritmo de *Strassen* teve desempenho inferior nas matrizes grandes, especialmente com *Pthreads*, limitado a 7 *threads*. Com *OpenMP* e *EIGEN*, o *Strassen* alcançou até 140x de *speedup*, mas ainda ficou abaixo do método trivial nas mesmas condições. Assim, o método trivial bem aprimorado mostrou-se mais eficiente e escalável para matrizes *4096x4096*.

CONSIDERAÇÕES FINAIS

Este trabalho apresentou uma análise aprofundada do desempenho de diferentes técnicas para a multiplicação de matrizes, considerando aspectos como paralelismo com *OpenMP* e *Pthreads*, técnicas de blocagem, vetorização com *SIMD(AVX)* e bibliotecas potencializadas como *BLAS* e *EIGEN*. A partir dos testes realizados, foi possível observar que a combinação dessas técnicas, especialmente com blocagem adequada e uso de bibliotecas potencializadas, resulta em ganhos expressivos de desempenho.

Dentre os algoritmos avaliados, o método trivial, quando associado às técnicas corretas, superou o algoritmo de *Strassen* na maioria dos casos,

principalmente para matrizes grandes. Ainda assim, *Strassen* demonstrou competitividade em casos específicos, como matrizes menores ou algumas combinações. A *API* de paralelização também influenciou os resultados, pois *OpenMP* apresentou maior praticidade de implementação, enquanto *Pthreads* se destacou pela estabilidade em configurações mais robustas.

Os resultados desta pesquisa reforçam estudos anteriores, como o de *Andrade e Cera* (1), ao comparar *OpenMP* e *Pthreads*, destacando que, embora o *OpenMP* seja mais simples e eficiente, o *Pthreads* pode alcançar melhor desempenho e estabilidade em cenários potencializados. Além disso, dialoga com a proposta de Oliveira *et al.* (12), ao mostrar que técnicas clássicas, como blocagem e vetorização, continuam sendo altamente eficazes quando bem ajustadas ao *hardware*, mesmo diante de novas abordagens paralelas.

Como continuidade deste trabalho, recomenda-se investigar o desempenho das técnicas em arquiteturas heterogêneas, especialmente com o uso de *GPUs* por meio de frameworks como *CUDA* e *OpenCL*, comparando os ganhos em relação às implementações em *CPU*. Também é interessante aplicar a metodologia a outros problemas computacionais, como sistemas esparsos, processamento de imagens e simulações numéricas. Além disso, pode-se explorar estratégias híbridas que combinam *CPU* e *GPU* de forma eficiente, potencializando o balanceamento de carga e reduzindo gargalos de comunicação.

Por fim, uma análise da eficiência energética das abordagens, especialmente em ambientes de computação embarcada e *edge computing*, pode oferecer uma compreensão mais aprofundada sobre os impactos das técnicas em termos de consumo e desempenho. O trabalho reforça a importância de conciliar fundamentos teóricos com experimentação prática para o desenvolvimento de soluções eficientes em computação paralela.

AGRADECIMENTOS

Agradeço ao Professor João Fabrício pela oportunidade e orientação em todas as etapas da pesquisa, à UTFPR pela infraestrutura oferecida, e ao CNPq pelo auxílio financeiro e concessão de bolsa - Projeto *01947 SISPEQ/UTFPR.*

REFERÊNCIAS

(1) *ANDRADE, Gabriella; CERA, Marcia. Comparação do uso de OpenMP e Pthreads em uma Paralelização de Multiplicação de Matrizes. In. [S. l.]: [S. n.], [S. d.].*

(2) *PACHECO, André G. C. Multiplicação de matrizes: uma comparação entre as abordagens sequencial (CPU) e paralela (GPU). CoRR, abs/1905.03641, 2019. arXiv: 1905.03641. Disponível em: http://arxiv.org/abs/1905.03641.*

(3) *STRASSEN, Volker. Gaussian Elimination is Not Optimal. Numerische Mathematik, Berlim, v. 13, p. 354–356, 1969. DOI: 10.1007/BF02165411.*

(4) *IEEE. POSIX Threads Programming. [S. l.], 2017. Disponível em: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html. Acesso em: 5 dez. 2023.*

(5) *GRAMA, Ananth et al. Introduction to Parallel Computing. 2. ed. [S. l.]: Addison-Wesley, 2003. ISBN 9780201648652.*

(6) *GUPTA, Anshul; KUMAR, Vipin. Scalability of Parallel Algorithms for Matrix Multiplication. [S. l.], 1991. Disponível em: https://www3.nd.edu/~zxu2/acms60212-40212-S12/scalability-of-paral lel-alg-for-matrix-multiplication.pdf. Acesso em: 5 dez. 2023.*

(7) *OPENMP ARCHITECTURE REVIEW BOARD. OpenMP Application Program Interface Version 5.1. [S. l.], 2020. Disponível em: https://www.openmp.org. Acesso em: 5 dez. 2023.*

(8) *GOTO, Kazushige; VAN DE GEIJN, Robert A. Anatomy of High-Performance Matrix Multiplication. ACM Transactions on Mathematical Software, New York, v. 34, n. 3, 12:1–12:25, 2008. DOI: 10.1145/1356052.1356053.*

(9) *INTEL CORPORATION. Details of Intel® Advanced Vector Extensions Intrinsics. [S. l.], 2011. Disponível em: http://portal.nacad.ufrj.br/online/intel/compiler_c/common/core/GUID-A9C3B12F-7A9A-4C8D-A6CD-9974ABC570E9.htm. Acesso em: 5 dez. 2023.*

(10) *LAWSON, Charles L. et al. Basic Linear Algebra Subprograms for Fortran Usage. ACM Transactions on Mathematical Software, v. 5, n. 3, p. 308–323, 1979. DOI: 10.1145/355841.355847*

(11) *GUENNEBAUD, GAËL AND JACOB, BENOÎT AND ET AL. Eigen: C++ Template Library for Linear Algebra. [S. l.], 2024. Versão 3.4.0. Disponível em: https://eigen.tuxfamily.org. Acesso em: 16 jul. 2025.*

(12) *OLIVEIRA, João, GONÇALVES, Rogério e FABRÍCIO FILHO, João. Multithread Approximation: A new OpenMP construct. In: ANAIS do XXV Simpósio em Sistemas Computacionais de Alto Desempenho. São Carlos/SP: SBC, 2024. p. 372–383. DOI: 10.5753/sscad.2024.244776. Disponível em: https://sol.sbc.org.br/index.php/sscad/article/view/31012.*

(13) *VIEIRA, Hiago Gimenez; GONÇALVES, Rogério Aparecido; FABRÍCIO FILHO, João. Desempenho da Multiplicação de Matrizes em Sistemas Multicore com Pthreads, OpenMP, Blocking, SIMD e BLAS. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DE SÃO PAULO (ERAD-SP), 16. , 2025, São José do Rio Preto/SP. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2025 . p. 38-41. DOI: https://doi.org/10.5753/eradsp.2025.9727.*