Performance evaluation of a single core and multi core

CDP Project

Turma 4 – Grupo 14

Gaspar Manuel Ferrás Faria (up202108797)

Miguel Filipe Rodrigues dos Santos (up202008450)

Rafael de Conceição Cerqueira (up201910200)

Licenciatura em Engenharia Informática e Computação

Índice

1.Introdução

Este projeto foca se na análise do desempenho de algoritmos de multiplicação de matrizes em diferentes ambientes de processamento, desde sistemas de núcleo único até implementações multi-core. O nosso objetivo é compreender como a hierarquia de memória afeta o desempenho do processador ao lidar com volumes significativos de dados. Para atingir este objetivo, iremos utilizar a API de Desempenho (PAPI) para recolher dados relevantes durante a execução dos programas. Através de uma análise abrangente, esperamos obter insights valiosos sobre a otimização de algoritmos para diferentes arquiteturas de processamento e compreender melhor as complexidades do desempenho computacional em ambientes de núcleo único e multi-core.

2.Multiplicação de Matrizes (Algoritmos)

Em busca de otimizar programas sequenciais, muitas vezes negligenciamos a importância da manipulação eficiente da memória. Neste projeto, exploramos a capacidade de aprimoramento de algoritmos sequenciais, propondo três abordagens distintas para medir o desempenho de um único núcleo diante de enormes volumes de dados, cada uma se diferenciando pela maneira como gerir alocação de memória. Esses algoritmos abrangem: Multiplicação Simples de Matrizes (já fornecido), Multiplicação de Matriz por Linha, Multiplicação de Matriz por Bloco.

Na tentativa de desenvolver soluções mais eficientes, encaramos o desafio de implementar códigos tanto em C/C++ quanto em uma linguagem adicional, optando pelo Java devido à sua afinidade com C/C++ em termos de abstração e sintaxe. Essa escolha não apenas facilitou a tradução de código entre as linguagens, mas também nos permitiu explorar as nuances de uma linguagem reconhecida por sua natureza híbrida entre interpretada e compilada, além de sua orientação totalmente voltada para objetos. Tais características têm potencial para influenciar significativamente o acesso à memória e, consequentemente, o tempo de execução dos algoritmos. Para a Multiplicação de Matriz por Bloco, porém, concentramo-nos exclusivamente na implementação em C/C++.

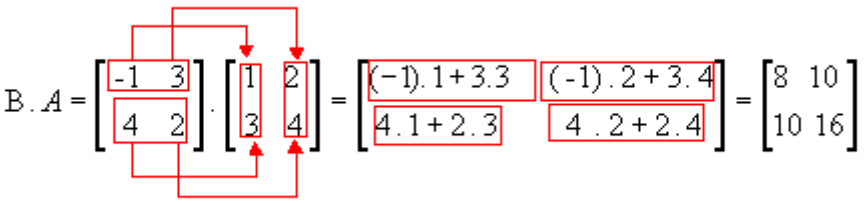
2.1 Multiplicação Simples de Matrizes

Neste método, cada valor presente na matriz A é multiplicado individualmente por cada elemento da linha correspondente da matriz B. Os resultados dessas multiplicações são então inseridos na linha apropriada da matriz resultante, denominada C.

$$A = \begin{bmatrix} 5 & 2 \\ 0 & -1 \\ 3 & 4 \end{bmatrix} \text{ e } B = \begin{bmatrix} -2 & 5 \\ 3 & 2 \\ 1 & 7 \end{bmatrix}$$

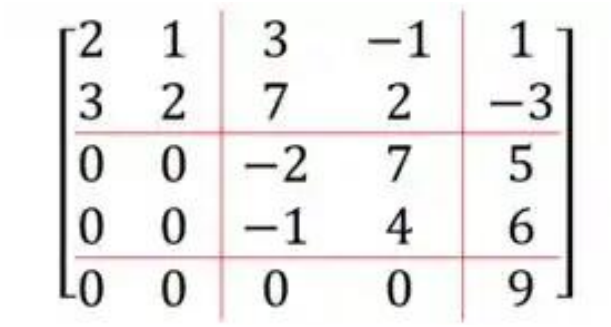
2.2 Multiplicação de Matrizes Linha a Linha

No algoritmo de multiplicação de matrizes linha-a-linha, cada elemento presente na matriz A é multiplicado individualmente por cada elemento da linha correspondente na matriz B. Os resultados dessas multiplicações são então inseridos na linha apropriada da matriz resultante C.



2.3Algoritmo de Multiplicação de Matrizes por Bloco

No caso deste método, desenvolvemos um algoritmo aprimorado que divide as matrizes originais em blocos de tamanho predefinido pelo utilizador. Em seguida, é realizada a multiplicação dos blocos correspondentes das duas matrizes para calcular a matriz resultante. Esta multiplicação entre os blocos é executada utilizando o algoritmo de multiplicação linha por linha.



3.Linguagens de programação testadas

Para avaliar o desempenho dos algoritmos nas versões em C/C++, utilizamos a Performance API (PAPI), que oferece acesso a diversas métricas da CPU e aos níveis de memória cache usados pelo processo. Além do tempo de execução do algoritmo, também consideramos o número de Operações de Ponto Flutuante (FLOP) e o número de falhas de cache, tanto para L1 quanto para L2. Falhas de cache podem resultar em um aumento significativo do tempo de processamento, por isso, é crucial monitorar sua variação para avaliar a eficiência da implementação. Esses valores não são estimados, são exatos e diretamente obtidos da API.

No presente estudo, a principal linguagem de programação utilizada foi o C++, escolhida devido à sua compatibilidade com a ferramenta PAPI. No entanto, para uma avaliação comparativa em diferentes ambientes, também decidimos utilizar Java. Uma característica importante do C++ é que ele armazena arrays bidimensionais em ordem de linha principal (row-major order), o que

significa que os elementos de uma matriz são armazenados em linhas contínuas na memória. Em contrapartida, em uma ordem de coluna principal (column-major order), os elementos das colunas são armazenados consecutivamente na memória.

Em ambas as linguagens, implementamos versões dos algoritmos de multiplicação, tanto por coluna quanto por linha, e analisamos o tempo de execução de cada uma. Para garantir resultados mais precisos, realizamos 10 execuções de cada algoritmo e calculamos a média dos resultados obtidos.

Os algoritmos implementados serão referidos como onMult, onMultLine e onMultBlock, correspondendo, respetivamente, aos algoritmos de multiplicação normal de matrizes, multiplicação linha-a-linha e multiplicação bloco-a-bloco.

3.1 Avaliação de Desempenho de Implementações Multi-Core para Produto de Matrizes

Na busca por melhorar a eficiência computacional, é fundamental explorar estratégias que tirem proveito das arquiteturas multi-core dos processadores modernos. Estas arquiteturas possuem múltiplos núcleos de processamento que podem executar tarefas simultaneamente, possibilitando um aumento significativo no desempenho computacional quando as aplicações são paralelizadas corretamente.

A multiplicação de matrizes é uma operação intensiva computacionalmente que pode se beneficiar enormemente da paralelização em sistemas multi-core. Ao dividir o trabalho em várias tarefas independentes e distribuí-las entre os núcleos do processador, é possível acelerar o cálculo do produto de matrizes e reduzir significativamente o tempo de execução. Ao analisar e comparar as implementações paralelas do algoritmo de produto de matrizes, é importante considerar como cada uma delas aproveita os recursos disponíveis nos sistemas multi-core. A eficácia da paralelização depende não apenas da divisão adequada do trabalho entre os núcleos, mas também da minimização de problemas como latência de comunicação entre os núcleos e contenção de recursos compartilhados.

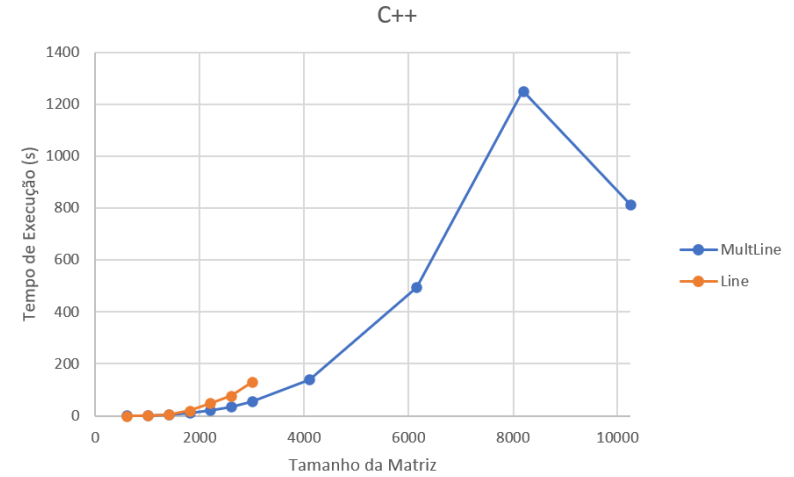
O processador Intel Core i7-8750H é uma unidade de processamento poderosa, com seis núcleos físicos e capacidade de processamento de até 12 threads simultâneas, graças à tecnologia Hyper-Threading. Com uma frequência máxima de 4,10 GHz, ele oferece um desempenho excecional para uma ampla variedade de cargas de trabalho computacionalmente intensivas, incluindo operações de multiplicação de matrizes. O processador possui uma arquitetura de cache eficiente, com 32 KB de cache L1 e 256 MB de cache L2. Esses caches ajudam a reduzir a latência de acesso aos dados e instruções, o que é crucial para melhorar o desempenho em tarefas que exigem acesso rápido à memória. Ao realizar a paralelização do algoritmo de multiplicação de matrizes no Intel Core i7-8750H, é essencial aproveitar ao máximo seus seis núcleos físicos e 12 threads. Distribuir adequadamente o trabalho entre esses núcleos, minimizando a latência de comunicação e evitando contenção de recursos compartilhados, é fundamental para obter o melhor desempenho possível. Além disso, é crucial monitorar métricas como o uso da CPU e a carga de trabalho em cada núcleo durante a execução das implementações paralelas. Isso permite identificar possíveis intervalos de desempenho e otimizar a distribuição de tarefas para maximizar a utilização dos recursos do sistema.

4.Resultados e analise

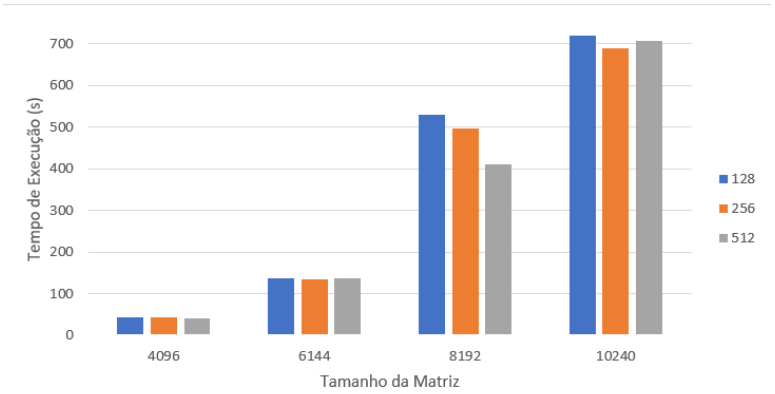
Para simplificar a compreensão dos dados, optamos por apresentá-los em gráficos de linhas e de barras. Esses gráficos foram criados utilizando as ferramentas disponíveis no Excel.

4.1 Tempo de execução

Os gráficos seguintes ilustram como o tempo de execução varia em relação ao tamanho da matriz para cada uma das linguagens de programação avaliadas: C++ e Java, em ambos os algoritmos "onmult" e "onMultLine". É notável que o tempo de execução permanece relativamente estável para cada linguagem de programação quando o tamanho da matriz está na faixa entre 600 e 3000. Contudo, ao testar tamanhos de matriz maiores que 3000 para o algoritmo "onMultLine" em C++, observou-se um crescimento exponencial no tempo de execução.



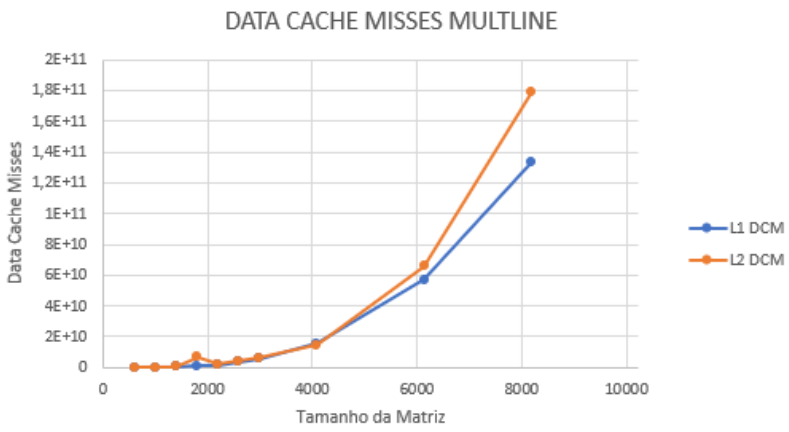
Quanto o algoritmo onMultBlock, introduzimos uma variável adicional: o tamanho dos blocos. Tal como observado anteriormente, o tamanho da matriz influencia significativamente o tempo de execução. No entanto, neste algoritmo, o tempo de execução também é afetado pelo tamanho dos blocos, pois diferentes divisões podem resultar em tempos de computação da multiplicação mais eficientes. Um tamanho de bloco muito pequeno pode levar a divisões da matriz original em submatrizes muito pequenas, aumentando o tempo necessário para carregar cada submatriz no processador. Por outro lado, se o tamanho do bloco for muito grande, o tempo de execução pode aumentar devido às grandes dimensões de cada submatriz a serem carregadas no processador.



Portanto, ao analisar os gráficos, podemos observar que o tamanho ideal do bloco seria 512, já que o algoritmo apresenta um tempo de execução menor para cada tamanho de matriz testado.

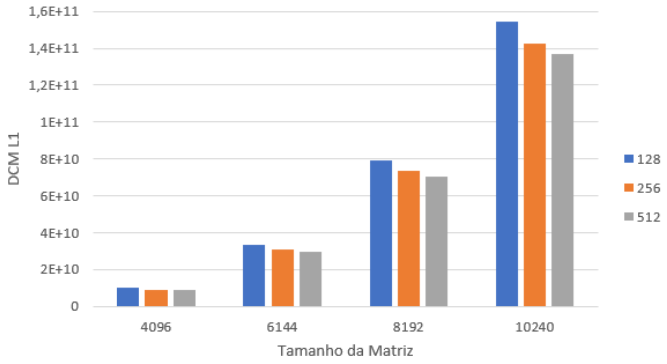
4.2 Data cache misses

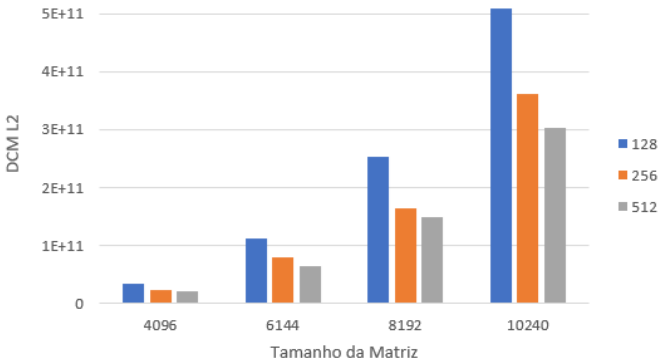
Quando o processador precisa aceder informações, o sistema primeiro verifica se essas informações estão disponíveis na cache. Se estiverem disponíveis, resulta em um "cache hit"; caso contrário, resulta em um "cache miss", o que significa que a informação precisa ser transferida da memória física (RAM) para a unidade de processamento. Cache hits permitem um acesso mais rápido às informações. No gráfico abaixo, observa-se o comportamento do número de falhas no acesso aos dados (cache misses) em relação ao tamanho das matrizes, especificamente para o algoritmo onMultLine:



Nos gráficos abaixo, é possível observar como o número de falhas no acesso aos dados nas memórias cache L1 e L2 variam conforme o tamanho da matriz e o tamanho do bloco utilizado. Essa relação está diretamente ligada ao volume de dados que precisam ser transferidos para o processador. Tanto o tamanho do bloco quanto o tamanho de cada matriz têm impacto no número de data cache misses, ou seja, na quantidade de vezes em que os dados precisam ser buscados na memória principal (RAM) devido à sua ausência na cache do processador.

Ao multiplicar matrizes por blocos, dividimos as operações de multiplicação em operações menores que cabem na memória cache. Isso reduz a quantidade de dados que precisam ser buscados na memória principal. Além disso, ao aumentarmos o tamanho do bloco, aumentamos a quantidade de dados que podem ser armazenados na cache. Portanto, quando utilizamos blocos maiores, é mais provável que mais dados necessários para as operações estejam na cache, diminuindo assim os data cache misses.





Esses gráficos nos permitem compreender como ajustar o tamanho do bloco pode influenciar a eficiência do algoritmo. Um tamanho de bloco adequado pode resultar em uma redução no número de cache misses e, consequentemente, em uma melhoria no desempenho computacional.

4.3 Abordagens Paralelas na Multiplicação de Matrizes

Nesta análise, exploramos duas abordagens distintas para a paralelização da multiplicação de matrizes, cada uma implementada em uma função separada. Ambas as funções empregam diferentes tipos de ciclos em sua estruturação para distribuir o trabalho entre as threads e aproveitar o potencial de processamento paralelo oferecido por sistemas multicore.

No primeiro ciclo paralelo, onde utilizamos a diretiva #pragma omp parallel for, o loop externo é distribuído entre as threads disponíveis. Cada iteração do loop externo é atribuída a uma thread separada, permitindo que várias threads processem diferentes partes do loop simultaneamente. Dentro do loop, as iterações internas também são executadas em paralelo por todas as threads, garantindo que o trabalho seja distribuído de maneira eficiente entre os núcleos do processador.

```
#pragma omp parallel for
for (int i=0; i<n; i++)
     for (int k=0; k<n; k++)
          for (int j=0; j<n; j++)
          {             }
```

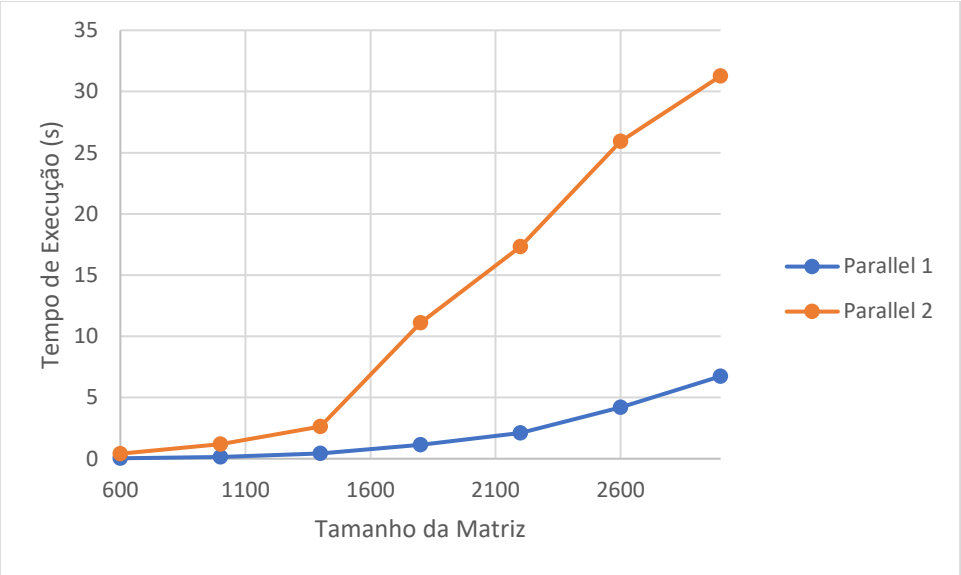
No segundo ciclo paralelo, onde utilizamos as diretivas #pragma omp parallel e #pragma omp for, o loop externo é executado por todas as threads disponíveis. Cada thread executa uma iteração do loop externo, garantindo que todas as iterações sejam processadas. Dentro do loop externo, as iterações internas são distribuídas entre as threads utilizando a diretiva #pragma omp for, o que significa que cada thread executa um subconjunto das iterações internas. Isso permite uma distribuição mais fina do trabalho entre as threads, reduzindo possíveis gargalos de desempenho e melhorando a eficiência do paralelismo.

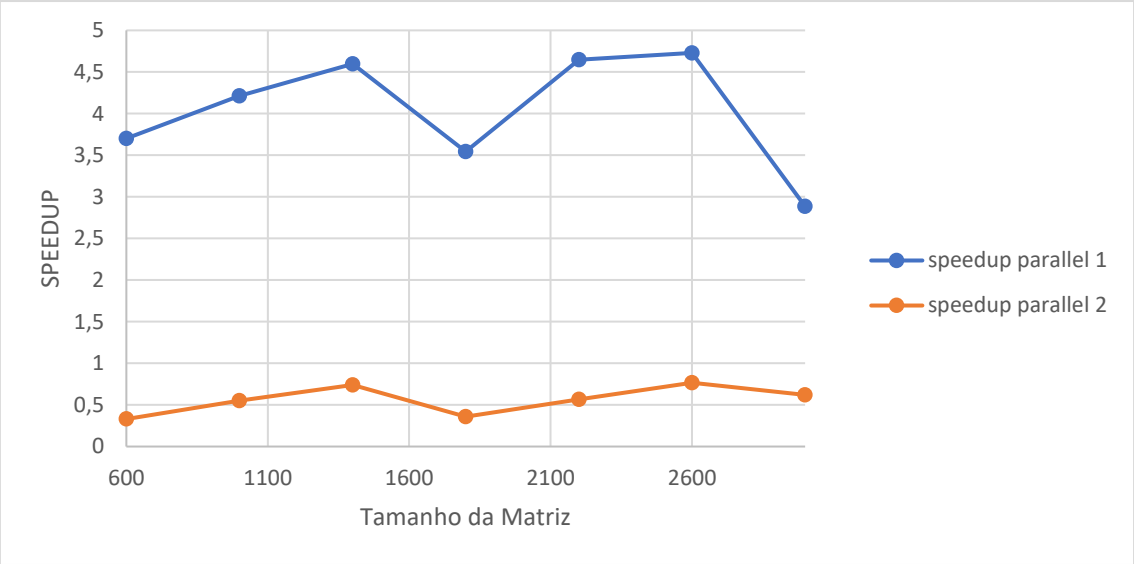
```
#pragma omp parallel
for (int i=0; i<n; i++)
     for (int k=0; k<n; k++)
          #pragma omp for
          for (int j=0; j<n; j++)
          {             }
```

A função multLineParallel1 utiliza a diretiva #pragma omp parallel for distribuindo todas as iterações do loop externo entre as threads criadas. Isso significa que cada iteração do loop externo é atribuída a uma thread separada, enquanto as iterações internas são executadas em paralelo por todas as threads. Enquanto a função multLineParallel2 utiliza a diretiva #pragma omp parallel para criar um grupo de threads, e em seguida utiliza #pragma omp for dentro do loop externo para distribuir as iterações internas entre as threads disponíveis. Isso significa que cada iteração do loop externo é executada por uma única thread, enquanto as iterações internas são distribuídas entre todas as threads.

Ao comparar o desempenho dessas duas abordagens, iremos analisar o tempo de execução de cada função, medido em segundos, e calcular o MFLOPS. Além disso, calcularemos o speedup, que é a relação entre o tempo de execução sequencial e o tempo de execução paralelo. Um speedup maior que 1 indica uma melhoria no desempenho com a paralelização. Por fim, calcularemos a eficiência, que é o speedup dividido pelo número de threads utilizados. Isso nos

dará uma medida de quão eficaz foi a utilização das threads disponíveis em relação ao desempenho obtido. Nos seguintes gráficos com essas análises, poderemos comparar e avaliar a eficácia de cada abordagem na paralelização da multiplicação de matrizes em termos de desempenho, utilização de recursos e eficiência.





Os gráficos demonstram que o a função com o apenas um #pragma omp for tem uma melhor performance que a segunda implementação, pois esta implementação divide todas as iterações i, j e k entre as threads disponíveis, logo cada thread executará todas as iterações dos loops internos resultando numa maior performance.

Para a segunda função funcionar tivemos que utilizar `#pragma omp parallel private(i,j,k),` pois serve para declarar as variáveis como privadas para cada thread impedindo conflitos de dados.

A eficiência (SPEEDUP/CORES) demonstra claramente que a primeira implementação é superior à segunda com os 6 cores disponíveis do processador utilizando as 12 threads.

5. Conclusão

Em síntese, este projeto nos permitiu explorar o impacto da hierarquia de memória em diferentes linguagens de programação, usando a multiplicação de matrizes como exemplo simples. Os resultados indicam que o algoritmo de multiplicação de matrizes por linha supera o método tradicional de multiplicação por coluna, aproveitando melhor a organização dos dados. Além disso, ao adotar a decomposição em blocos, observamos uma melhoria adicional no desempenho, especialmente quando o tamanho dos blocos foi criteriosamente escolhido. Isso resultou em uma melhor utilização da memória cache, reduzindo os misses e aprimorando a eficiência do algoritmo. Essas descobertas ressaltam a importância de considerar a hierarquia de memória ao projetar algoritmos e programas, visando otimizar o desempenho e a eficiência nos sistemas computacionais.