Faculdade de Engenharia da Universidade do Porto

Projeto 1 CPD

Avaliação de Performance do CPU em single-core e multi-core

Turma 3 - Grupo 19

João Brandão Alves - up202108670@up.pt
Eduardo Machado Teixeira de Sousa - up202103342@up.pt

Índice

Avaliação de performance com single-core e multi-core

O projeto proposto serve para mostrar o efeito na performance do processador devido à hierarquia da memória quando é necessário obter acesso a grandes quantidades de informação. Além disso, o projeto também serve para mostrar que um processo pode ser realizado mais rapidamente usando vários threads do processador, mas nem sempre compensa devido aos recursos gastos em relação à diminuição do tempo proporcionada.

Descrição do Problema

A nossa tarefa neste projeto é utilizar o produto de duas matrizes (que por natureza exige muitos recursos computacionais para realizar) para estudar os efeitos de aceder a grandes quantidades de entradas de dados na performance do processador.

Explicação dos Algoritmos

Para este trabalho, desenvolvemos 5 algoritmos diferentes para resolver o problema de multiplicação entre matrizes. Os 3 primeiros apresentados, que diferem na manipulação de memória, têm como objetivo medir a performance de um único thread do processador face a grandes grupos de dados.

Os 2 últimos algoritmos têm como objetivo medir a performance de vários threads do processador aquando de executarem tarefas para um mesmo processo em paralelo, sendo que este mesmo processo requer uma grande quantidade de dados. Estes 2 últimos têm apenas uma mudança que é onde é que os processos irão ser separados para serem executados em paralelo.

Multiplicação Simples

Para o primeiro algoritmo era-nos dado no enunciado código em *c/c++*, com complexidade temporal $O(n^3)$, para multiplicar duas matrizes multiplicando sequencialmente cada linha da primeira matriz por cada coluna da segunda. **Pseudo código:**

```
for(int i=0; i<m_ar; i++)
     for(int k=0; k<m_ar; k++)
          temp = 0;
          for(int j=0; j<m_br; j++)
               temp += pha[i*m_ar+k] * phb[k*m_br+j];
          phc[i*m_ar+j]=temp;
```

Multiplicação por Linhas

No segundo algoritmo desenvolvemos código (com a mesma complexidade temporal) que multiplica cada elemento da primeira matriz pela linha da segunda matriz que lhe corresponde e vai construindo a matriz resultante. Com este algoritmo o tempo de execução foi bastante mais reduzido do que quando comparado à execução do algoritmo anterior. **Pseudo código:**

```
for(int i=0; i<m_ar; i++)
     for(int k=0; k<m_ar; k++)
          for(int j=0; j<m_br; j++)
               phc[i*m_ar+j]+= pha[i*m_ar+k] * phb[k*m_br+j];
```

Multiplicação por Blocos

Finalmente, o terceiro método consiste em dividir as matrizes em matrizes mais pequenas, realizar o cálculo nessas matrizes e depois somar tudo para obter o resultado final. Embora a complexidade temporal se mantenha em $O(n^3)$, a ideia é que existirá mais aproveitamento das caches, limitando o acesso a memórias mais lentas.
Pseudo código:

```
for(i = 0; i < m_ar; i += blockSize)
     for(j = 0; j < m_ar; j += blockSize)
          for(k = 0; k < m_br; k += blockSize)
               for (x = i; x < i + blockSize; x++)
                    for (y = j; y < j + blockSize; y++)
                         for (z = k; z < k + blockSize; z++)
                              phc[x*m_ar+z] += pha[x*m_ar+y] * phb[y*m_br+z];
```

Multiplicação por Linhas em Paralelo

Neste código, a tag *#pragma omp parallel* for é aplicada apenas ao *loop* mais externo (o *loop* que itera sobre i). Esta diretiva diz ao compilador para executar este ciclo em paralelo, distribuindo as iterações do ciclo i por diferentes threads. Os loops internos são executados totalmente por cada thread que executa uma parte do *loop* i. Esta abordagem é simples, mas pode nem sempre levar ao melhor desempenho, dependendo da carga de trabalho dentro do loop. **Pseudo código:**

```
#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++)
              {             }
```

Esta abordagem envolve a *tag #pragma omp parallel* aplicada ao bloco for mais externo e uma *tag #pragma omp for* apenas ao *for loop* mais interno. Este método tenta aproveitar o paralelismo em dois níveis: o paralelismo entre as linhas da matriz é explorado pelo *loop* externo, onde diferentes *threads* podem trabalhar em diferentes linhas simultaneamente; e o paralelismo dentro de uma linha específica da matriz é explorado pelo paralelismo no loop mais interno, permitindo que múltiplos *threads* colaborem no cálculo de uma única linha. **Pseudo código:**

```
#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++)
               {             }
```

Métricas de Performance

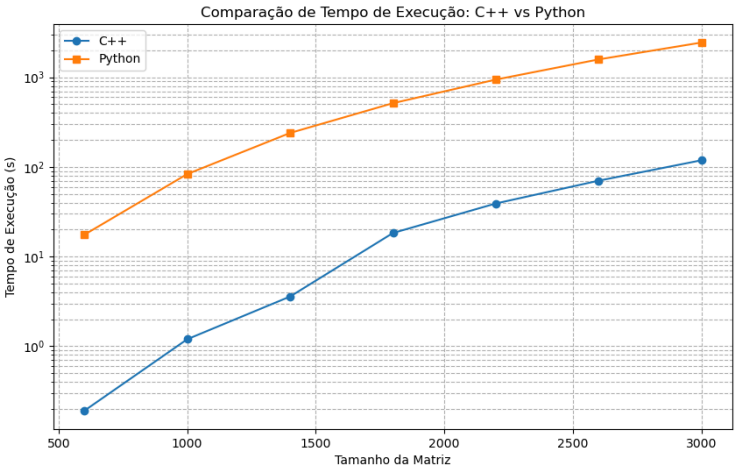
Para avaliar a performance dos algoritmos, utilizamos a API de desempenho (PAPI) para medir a atividade da CPU e a utilização da memória cache com precisão, com intuito de avaliar o desempenho dos algoritmos em *c/c++*. Como têm um efeito na eficiência, as principais medidas foram os erros de cache nos níveis L1 e L2, os tempos de execução e as operações de ponto flutuante, estas últimas apenas para os casos de execução paralela. As medições foram efectuadas numa única máquina da FEUP com sistema Ubuntu 22.04, com um CPU

i7 9700 e foram obtidas diretamente, sem quaisquer aproximações. A fiabilidade dos dados foi garantida executando os testes pelo menos 3 vezes para os casos de algoritmos em *c/c++* e 2 vezes para os casos de algoritmos em python, e calculando a média dos resultados para reduzir as oscilações. Para evitar a sobreposição de alocação de memória, um novo processo foi iniciado para cada teste. Foi usada a *flag* de otimização -O2 para compilar os programas *c/c++*, tal como pedido. A avaliação de performance também contrasta esses resultados com os mesmos algoritmos implementados em *python*.

Resultados e Análises

Multiplicação Simples

Observa-se que o código em *python* demora consideravelmente mais tempo que em *c++*, especialmente com tamanhos de matriz pequenos, onde fica duas ordens de grandeza acima do c++, devido à baixa proximidade que tem ao *hardware* em relação ao código de *c++*. Observa-se também que à medida que aumenta o tamanho da matriz, a diferença de tempo entre as duas linguagens diminui consideravelmente, sendo de apenas uma ordem de grandeza quando as matrizes são de 3000x3000, ainda que o código em *python* se mantenha consideravelmente mais lento que o código em *c++*. Deste modo, é impensável usar a implementação em python devido ao seu tempo de execução para tarefas relativamente pequenas. A única maneira de otimizar o código *python* seria usar a biblioteca do *numpy*, o que basicamente iria converter este código para *c/c++*, otimizando-o assim.



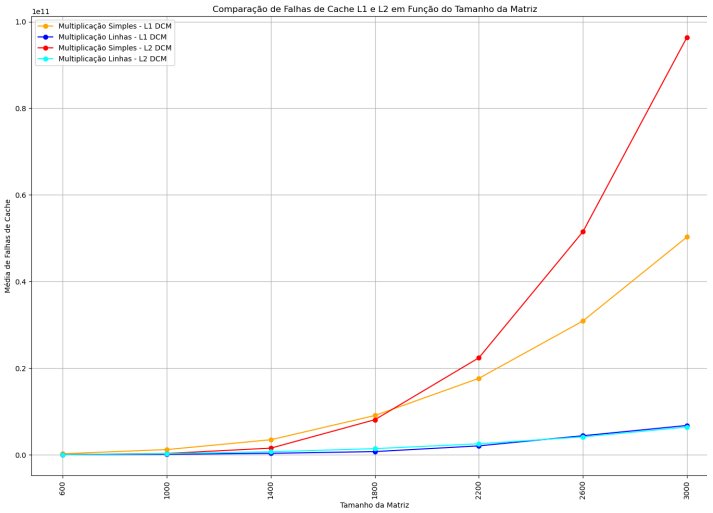
Multiplicação por Linhas

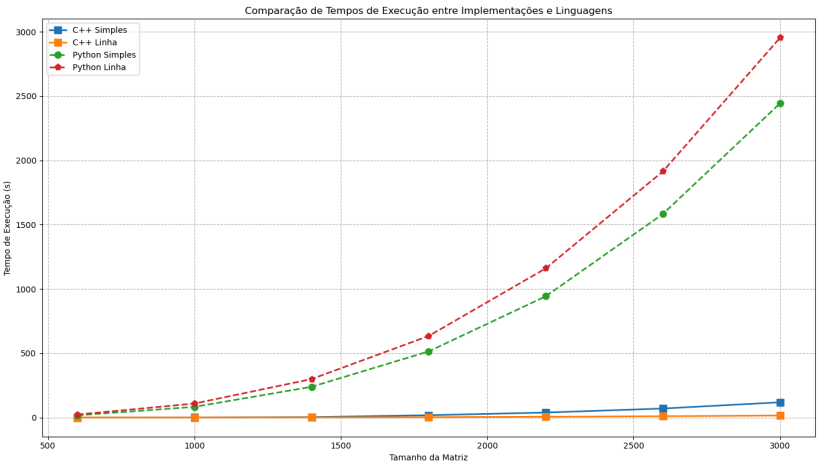
Observamos que as duas linguagens diferem bastante no tempo de execução, sendo que em Python este é exponencialmente maior. Ao comparar os dois algoritmos (Multiplicação Simples e Multiplicação por Linhas), observamos que em *c++* este último é mais eficiente, o que é mais evidente quanto maior o tamanho da matriz (sendo que a 3000x3000, o primeiro algoritmo demora cerca de 10 vezes mais). No entanto, esta relação não se verificou no *python*, na qual os dois algoritmos têm performances semelhantes (sendo o segundo ligeiramente menos eficiente).

Esta diferença de execução temporal marcada entre as duas linguagens deve-se a estas seguirem paradigmas diferentes, sendo que *c++* é compilada e *python* é interpretada. Isto leva a que o que *python* seja uma linguagem de nível mais alto, diminuindo assim a proximidade com o *hardware*, o que leva, consequentemente, a que as operações e acessos envolvendo todo o tipo de memórias sejam consideravelmente mais lentas quando comparado com uma linguagem como *c++*.

A eficiência superior do código que faz operações de multiplicação linha por linha, comparativamente ao código de multiplicação simples, pode ser atribuída à redução significativa nas falhas de cache durante a execução. Essa melhoria ocorre porque, na multiplicação simples, que processa os elementos linha por coluna, quase todas as operações resultam em falhas de cache. Isso deve-se ao fato de que, após uma falha de cache, apenas um pequeno bloco de dados (aproximadamente equivalente a uma linha da matriz e o número utilizado na

multiplicação corrente) é puxado da memória principal. Consequentemente, na operação subsequente, o algoritmo de multiplicação simples precisa de pedir o próximo número da coluna da memória, visto que ele não está disponível no cache. Por outro lado, ao executar as multiplicações linha por linha, o algoritmo de multiplicação por linhas aproveita eficientemente o bloco de dados já presente na cache, resultado da falha anterior. Essa abordagem minimiza as operações de acesso à memória, resultando numa boa redução do tempo de execução em comparação com o algoritmo de multiplicação simples, que não otimiza o uso do cache de forma tão eficaz.



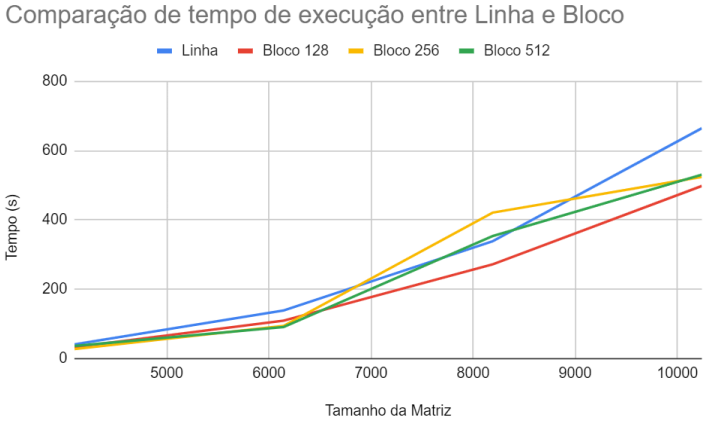


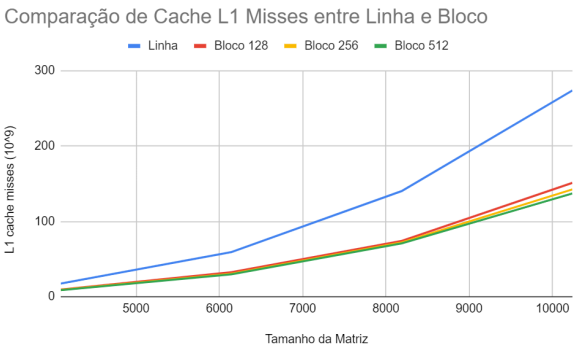
Multiplicação por Blocos

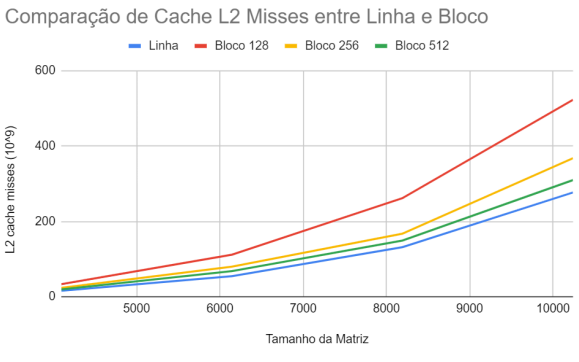
Ao comparar o tempo demorado pelo algoritmo anterior com várias instâncias da Multiplicação por Blocos (blocos de 128, 256 e 512 respectivamente), observa-se que tendencialmente este último é mais rápido, embora não por uma diferença muito significativa. Também se nota uma tendência de mais rapidez com blocos maiores, no entanto a inconsistência dos dados obtidos ao nível dos tempos não nos permite tirar as conclusões que deveriam ser observáveis. Isto deve-se, muito provavelmente, a uma implementação mal otimizada da ideia do algoritmo da multiplicação por blocos.

Ao analisar os dados de falhas de cache dos dois métodos, notamos que no que toca à cache L1, a Multiplicação por Linhas excede largamente o número de falhas de cache dos três métodos por blocos. No entanto, nas falhas de cache em L2, constatamos o inverso: a multiplicação por linhas tem menos falhas de cache que a multiplicação por blocos. Isto acontece porque quando existe uma falha de um valor em cache, o método dos blocos vai buscar uma grande quantidade de dados sequencialmente à memória, dados estes que irão ser usados em operações num futuro não muito distante ao momento em que a execução se encontra naquele momento. Deste modo, o número de vezes que a execução do código tem que recorrer à memória principal é mais reduzido, o que faria com o tempo de execução fosse inferior do que quando comparado ao método de multiplicação por linhas (o que infelizmente não se constatou nos dados que obtemos).

Com o aumento do tamanho dos blocos, a média das falhas de cache diminuiu um pouco, o que mostra que se cada vez que a execução programa precisar de ir à memória principal por falta de dados, para os colocar em cache, compensa ir buscar mais informação de uma vez só, pois essa informação que o bloco contém irá, provavelmente, ser usada quase de seguida para efetuar mais operações. Com a diminuição das idas à memória para buscar dados, seria de esperar que o tempo de execução também seguisse a mesma tendência porque as operações mais demoradas (que são as idas à memória) foram reduzidas mas, infelizmente, não pudemos concluir isso através da nossa recolha de dados provavelmente, devido à má otimização na implementação do algoritmo.







Multiplicação por Linhas em Paralelo

Na análise dos dados obtidos a partir de 3 medições para cada caso, identificamos diferenças significativas entre os dois algoritmos de multiplicação de matrizes em paralelo.

O primeiro algoritmo, caracterizado pela utilização de 1 única *tag #pragma*, demonstrou ser superior em diversos aspectos, incluindo tempos de execução, número de operações de ponto flutuante por segundo (*flops*) e ganho de desempenho (*speedup*), em comparação ao segundo algoritmo, que se distinguia pela aplicação de 2 *tags pragma*. No entanto, observou-se que, apesar de sua superioridade evidente, o primeiro algoritmo começou a apresentar uma utilização menos eficiente dos recursos disponíveis conforme o tamanho das matrizes aumentava.

Comparando com o algoritmo de multiplicação de matrizes sem paralelização, o primeiro algoritmo paralelizado justifica amplamente o uso de mais recursos computacionais, oferecendo uma solução significativamente mais rápida para resolver o problema. Por outro lado, o segundo algoritmo paralelizado não demonstra uma justificativa clara para o uso intensivo de recursos computacionais, visto que seus resultados não superam substancialmente a versão que funciona com um único *thread*.

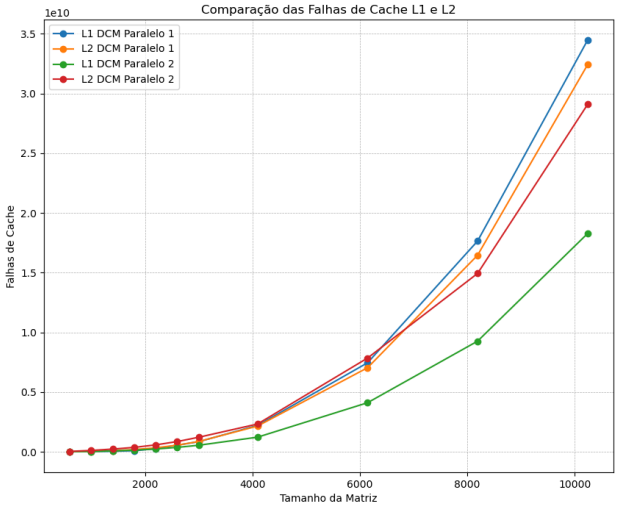
Em termos de desempenho, a escolha da melhor abordagem depende um pouco da arquitetura do sistema e do tamanho da matriz a ser processada. Se o sistema tiver uma grande quantidade de memória cache e for capaz de lidar eficientemente com acessos à memória não sequenciais, o primeiro algoritmo paralelo é preferível. No entanto, se o sistema tiver uma quantidade limitada de memória cache ou se o tamanho da matriz for muito grande, a segunda abordagem pode ser mais eficiente, pois pode resultar em menos falhas de cache. Ainda assim, para matrizes demasiado grandes a implementação do algoritmo deveria ser diferente das 2 opções apresentadas, de forma a que fosse possível fazer um melhor uso das memórias cache e assim reduzir ainda mais o tempo de execução.

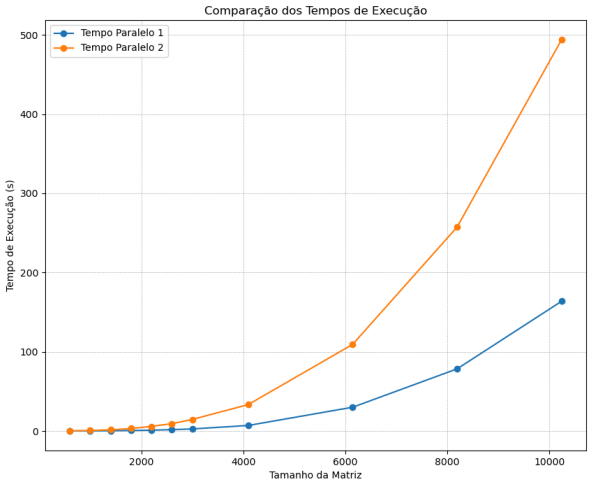
Além disso, enfrentamos um problema inicialmente com o segundo algoritmo: embora executasse normalmente, não realizava as multiplicações matriciais esperadas, resultando numa execução infinita. A causa do problema foi identificada como a omissão da declaração *int* nos ciclos *for*, o que impedia que as variáveis dos ciclos fossem tratadas como locais. Assim, quando as tarefas eram distribuídas paralelamente entre os diversos processos, a ausência de variáveis locais significava que o fragmento de código executado em cada processo não tinha acesso às variáveis necessárias para realizar as multiplicações matriciais, levando assim a que a execução do código não acabasse. Este problema que encontramos destaca a importância de uma verificação cuidadosa das implementações em ambientes de computação paralela, onde detalhes simples, como a declaração de variáveis, podem ter impactos significativos na execução dos algoritmos.

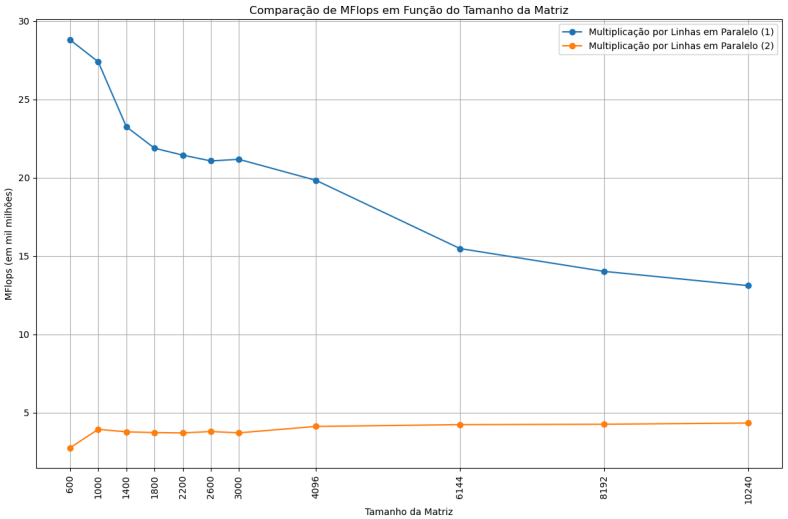
Inicialmente, a expectativa era de que a versão do algoritmo com duas tags *#pragma* superasse a versão com uma única *tag* em termos de desempenho, devido à sua capacidade de distribuir um maior número de tarefas simultaneamente. No entanto, esta hipótese não se confirmou nas medições de *flops* e de *speedup*, o que resultou em uma eficiência consideravelmente reduzida. Uma possível explicação para isto pode ser a excessiva paralelização. Isso pode ter levado a uma situação em que tarefas foram atribuídas a *threads* que já se encontravam sobrecarregadas com outras tarefas. Mesmo que essas *threads* não necessitassem realizar novos acessos à memória - devido à disponibilidade dos dados necessários em cache - isso não se traduziu numa vantagem significativa. De fato, observou-se que a versão com duas *tag #pragma* apresentou um número menor de falhas de cache em comparação à versão com uma única *tag*. Este resultado sugere uma complexa interação

entre paralelização, gestão de tarefas e eficiência no acesso à memória, reforçando a necessidade de haver equilíbrio na distribuição de tarefas entre os *threads* disponíveis para otimizar o desempenho do algoritmo.

Num último teste, mais por alto, de performance, verificamos que ao reduzir o número de threads usados na paralelização também pode influenciar bastante nos resultados obtidos, pois a medições apresentadas no nosso relatórios foram todas baseadas no uso de 8 *threads* (todos os *threads* disponíveis no processador) mas, quando fizemos alguns testes com apenas 6 *threads* verificamos que obtemos melhores resultados usando menos recursos. Isto deve-se provavelmente ao facto de alguns *threads* já estarem ocupados a executar outras tarefas (como correr o sistema operativo), o que nos leva a concluir que nem sempre a melhor solução é usar todos os recursos disponíveis e sim, fazer antes uma avaliação mais simples no início para concluir a melhor quantidade de recursos a serem usados na paralelização.







Conclusões

Este projeto abordou o impacto da hierarquia de memória na eficiência do processador, centrando-se na multiplicação de matrizes. Analisamos o desempenho de um único *thread* utilizando diferentes linguagens de programação e algoritmos, incluindo uma comparação entre a multiplicação simples, a multiplicação de elementos em linha e estratégias orientadas com blocos. Além disso, exploramos melhorias de desempenho multi-core através da paralelização com o OpenMP. As nossas conclusões sublinham o papel crucial dos padrões de acesso à memória e da localidade dos dados na otimização do desempenho, especialmente para grandes conjuntos de dados. A abordagem orientada por blocos melhorou a eficiência ao reduzir as falhas de cache, enquanto as estratégias de paralelização ofereceram aumentos de velocidade, destacando o potencial da otimização personalizada no aproveitamento de arquitecturas multi-core para tarefas computacionais.