Faculdade de Engenharia

da Universidade do Porto

Computação paralela e distribuída

Primeiro Trabalho

2024/2025 – Licenciatura em Engenharia Informática e Computação

Autores:

Pedro Marinho up202206854@up.pt

Sérgio Nossa up202206856@up.pt

Xavier Martins up202206632@up.pt

Índice

Introdução

Neste trabalho, avaliamos o desempenho de vários algoritmos criados de forma a comparar métodos de otimização da cache na multiplicação de matrizes em várias linguagens, bem como o desempenho de versões multi-threaded destes algoritmos.

Implementação dos algoritmos

No âmbito deste projeto, desenvolvemos e analisamos três algoritmos distintos de multiplicação de matrizes:

- Multiplicação Matricial Convencional
- Multiplicação Matricial Linha por Linha
- Multiplicação Matricial por Blocos

Estes algoritmos distinguem-se pela forma como acedem sequencialmente aos elementos na memória principal, resultando em diferentes níveis de eficiência na utilização da cache e, consequentemente, no desempenho global. No caso específico do terceiro algoritmo, utilizamos diferentes tamanhos de blocos para avaliar o seu impacto no desempenho.

De modo a realizar uma análise comparativa mais abrangente, implementámos os dois primeiros algoritmos em C++, Java, GO e Python, permitindo avaliar o impacto da linguagem de programação no desempenho computacional.

Adicionalmente, exploramos a paralelização através da implementação do primeiro e segundo algoritmo em versão single-core e multi-core. A versão multi-core foi desenvolvida seguindo duas abordagens distintas e testada com diferentes configurações de threads, possibilitando uma avaliação detalhada das vantagens da computação paralela neste contexto.

Multiplicação de Matrizes Convencional

Este algoritmo implementa a abordagem clássica da multiplicação de matrizes: iteramos por cada elemento da matriz resultante, calculando o produto escalar entre a linha correspondente da primeira matriz e a coluna correspondente da segunda matriz. Para cada posição (i,j) da matriz resultante, percorremos sistematicamente os índices k da primeira linha e da segunda coluna, acumulando o somatório dos produtos $a_{i\square} \times b_{\square\square}$.

A definição matemática que representa esta operação é expressa pela fórmula:

$$c_{i\square} = \sum_{k=1}^{m} a_{i\square}\, b_{\square\square}$$

Onde c representa a matriz resultante, a e b são as matrizes a multiplicar, e m corresponde à dimensão interna compatível entre ambas (número de colunas de a e número de linhas de b). Para visualizar o pseudocódigo.

A análise de desempenho deste algoritmo revela uma eficiência inferior quando comparado com as outras implementações desenvolvidas. Esta ineficiência deve-se principalmente ao padrão de acesso à memória, que não otimiza a utilização da hierarquia de cache. A necessidade de aceder repetidamente às mesmas colunas da segunda matriz para diferentes linhas da primeira matriz provoca um elevado número de *cache misses*, reduzindo significativamente a velocidade de execução devido às constantes transferências de dados entre a memória principal e a cache.

Multiplicação Matricial Linha por Linha

Este algoritmo apresenta uma otimização significativa relativamente à abordagem convencional. Em vez de percorrer sequencialmente cada coluna da segunda matriz para cada linha da primeira, esta implementação reorganiza o acesso aos dados, processando as matrizes linha a linha.

A principal vantagem desta abordagem reside na forma como explora eficientemente a localidade espacial da cache. Quando acedemos a um elemento na memória principal, o sistema carrega automaticamente para a cache não apenas esse elemento, mas também um bloco contíguo de dados adjacentes. Assim, ao percorrer as matrizes sequencialmente por linhas, maximizamos a probabilidade de que os próximos elementos necessários já se encontrem em cache, reduzindo significativamente o número de acessos à memória principal.

Esta reorganização do padrão de acesso à memória permite uma utilização mais eficiente do subsistema de memória, particularmente nas arquiteturas modernas que implementam múltiplos níveis de cache. O algoritmo mantém-se matematicamente equivalente ao convencional, produzindo resultados idênticos, mas com um tempo de execução consideravelmente inferior devido à melhor utilização dos recursos de hardware disponíveis.

As medições de desempenho comprovam esta otimização, revelando uma redução substancial no tempo de processamento, especialmente para matrizes de grandes dimensões, onde os efeitos da localidade de cache se tornam ainda mais pronunciados. Para visualizar o pseudocódigo.

Multiplicação Matricial por Blocos

Este algoritmo representa uma evolução sofisticada das abordagens anteriores, implementando uma estratégia de divisão e conquista para otimizar o desempenho. A técnica consiste em decompor as matrizes originais A e B em sub-blocos menores, que são então multiplicados individualmente.

A implementação permite duas variantes distintas para a multiplicação dos blocos: podemos aplicar tanto o algoritmo "Linha por Linha" como o algoritmo "Convencional" a cada par de blocos correspondentes. Esta flexibilidade possibilita uma análise comparativa do desempenho de ambas as abordagens no contexto da multiplicação por blocos.

A principal vantagem desta decomposição em blocos reside na otimização simultânea de dois aspectos críticos da hierarquia de memória: a localidade espacial e a localidade temporal da cache. Por um lado, ao processar os dados em blocos contíguos, maximizamos o aproveitamento da localidade espacial, já que elementos vizinhos são carregados conjuntamente na cache. Por outro lado, ao completar todas as operações necessárias com um bloco antes de prosseguir para outro, aumentamos significativamente a reutilização dos dados na cache (localidade temporal), uma vez que os mesmos elementos são acedidos repetidamente dentro de um curto intervalo de tempo.

Multiplicação Matricial Linha a Linha e Convencional com Multi-core

Este algoritmo representa um avanço significativo ao explorar o paradigma de computação paralela para otimizar a multiplicação de matrizes, utilizando a biblioteca OpenMP para distribuir eficientemente o processamento entre múltiplos núcleos do processador. A implementação foi desenvolvida em duas versões distintas, cada uma adotando uma estratégia diferente de paralelização:

Versão 1 - Paralelização do Ciclo Externo

Nesta primeira abordagem, a paralelização é aplicada ao ciclo mais externo do algoritmo. As threads são criadas e distribuídas para executar diferentes iterações deste ciclo principal, dividindo efetivamente as linhas da matriz resultante entre os núcleos disponíveis. Cada thread processa independentemente um subconjunto das iterações totais, o que resulta numa distribuição equilibrada da carga de trabalho pelos diversos núcleos do processador.

Esta estratégia minimiza a sobrecarga de sincronização, uma vez que cada thread pode operar de forma relativamente independente na sua porção designada dos dados, com pouca necessidade de comunicação entre threads durante o processo de cálculo.

Versão 2 - Paralelização do Ciclo Interno

A segunda versão implementa uma região paralela utilizando a diretiva #pragma omp parallel, onde cada thread executa o código contido nessa região. No entanto, a paralelização é especificamente aplicada ao terceiro ciclo através da diretiva #pragma omp for. Nesta configuração, cada thread executa independentemente os dois primeiros ciclos e, posteriormente, sincroniza-se no final do terceiro ciclo para a multiplicação e acumulação paralela dos resultados para cada coluna da matriz b.

Esta abordagem permite um maior grau de paralelismo ao nível mais granular do algoritmo, potencialmente aproveitando melhor as capacidades de processamento paralelo em determinados cenários ou arquiteturas específicas.

Ambas as versões foram implementadas tanto para o algoritmo de multiplicação Linha a Linha (V1-V2) como para o algoritmo Convencional (V1-V2), permitindo uma análise

comparativa abrangente das diferentes combinações de estratégias de acesso à memória e técnicas de paralelização.

Metodologia de Avaliação

A avaliação do desempenho dos nossos algoritmos baseiam-se em diversos indicadores-chave, com cada um a mostrar perspectivas valiosas sobre diferentes aspetos da eficiência computacional:

1. **Tempo de Execução**: Este indicador mede a duração necessária para completar o algoritmo de multiplicação de matrizes. Um tempo de execução mais curto indica melhor desempenho.
2. **Falhas de Cache**: Na implementação em C/C++, utilizámos a Performance API (PAPI) para monitorizar o número de falhas de cache ao nível 1 (L1 Data Cache Misses, L1_DCM) e nível 2 (L2 Data Cache Misses, L2_DCM). As falhas de cache são um indicador crítico, pois influenciam significativamente o tempo de processamento. Uma falha de cache ocorre quando a CPU não consegue localizar os dados necessários na memória cache, levando a uma recuperação mais demorada a partir da memória principal ou da cache de nível inferior. Este indicador é particularmente relevante na comparação da eficácia de diferentes abordagens algorítmicas.
3. **Milhões de Operações de Ponto Flutuante por Segundo (MFLOPS)**: Calcula-mos os MFLOPS para aferir a potência de processamento bruta da nossa implementação. Antecipamos variações neste indicador entre os diferentes algoritmos de multiplicação de matrizes, com valores mais elevados a indicar um desempenho computacional superior.
4. **Aceleração e Eficiência**: Para cada execução dos algoritmos paralelizados, calculamos a aceleração e a eficiência em comparação com a implementação single-core.

Notas adicionais:

- Todos os testes foram realizados utilizando um processador AMD Ryzen 7 7840 HS com 8 cores e 2 threads por core, operando num ambiente WSL 2 em Windows 11.
- Para garantir a precisão dos nossos dados, cada teste foi repetido pelo menos 30 vezes no mesmo sistema, de forma a garantir maior certidão dos resultados.
- As implementações em Java e Go não incluem a métrica de falhas de cache na nossa análise.
- A gama de tamanhos de matriz testados estendeu-se de 600x600 até 10240x10240. Foram testados blocos de tamanho 128, 256 e 512 para o algoritmo de multiplicação orientado a blocos.
- 3 resultados dos testes de multiplicação por blocos não são considerados na análise de dados, pois apresentam valores 10x acima do esperado.
- Para os testes de paralelização, utilizamos diferentes configurações de threads: 4, 8, 12 e 24.
- Para a versão C++, foi utilizada a flag de otimização -O2 na compilação.

- O tempo de execução dos algoritmos em Python revelou-se excessivamente elevado, o que impossibilitou a realização dos testes dentro de um prazo viável. Por esse motivo, os resultados de desempenho em Python não serão analisados na próxima secção.

Análise de dados

Comparação do tempo de execução para tamanhos de blocos diferentes no algoritmo de multiplicação por blocos

Durante os testes, reparamos que o crescimento do tempo de execução do algoritmo de multiplicação por blocos convencional é significativamente superior ao do algoritmo de multiplicação por linha, dificultando o procedimento dos testes. Sendo assim, só analisamos o algoritmo com matrizes de tamanho até 3000x3000.

Observando os gráficos, podemos concluir que diferentes tamanhos de blocos maximizam o desempenho do algoritmo consoante o tamanho da matriz, 128 sendo o melhor tamanho para matrizes de tamanho inferior ou igual a 3000, 512 para matrizes de tamanho entre 4096 a 6144 e 256 para matrizes de tamanho superior a 6144.

Comparação do desempenho e optimização da utilização da cache dos algoritmos implementados

Comparando os algoritmos, podemos observar que, como esperado, a variação da multiplicação por blocos com multiplicação linha por linha minimiza o número de *cache misses*. O algoritmo de multiplicação linha por linha apresenta resultados próximos, com com número superior de *cache misses* na cache L1 e número inferior na cache L2.

Observando o tempo de execução, podemos concluir que os algoritmos que otimizam a utilização da cache têm um menor tempo de execução. Podemos assim concluir que o acesso rápido à memória que a cache permite contribui significativamente para o desempenho destes algoritmos.

Comparação do desempenho de linguagens de programação diferentes na multiplicação de matrizes

Ao analisar os tempos de execução do algoritmo convencional e do algoritmo de multiplicação linha por linha, observa-se que Java e Go apresentam desempenhos semelhantes. No entanto, o código em C++ só se mostra mais rápido que as outras linguagens quando é utilizado o algoritmo linha por linha.

Como o principal *bottleneck* no desempenho desses algoritmos está na latência de acesso à memória cache, quando não há uma otimização voltada para a utilização eficiente da cache, as vantagens de desempenho que o C++ normalmente oferece em termos de velocidade não se manifestam de forma significativa.

Comparação do desempenho de algoritmos multi-core para multiplicação de matrizes

A partir dos gráficos para o algoritmo V1, observamos que um maior número de threads geralmente resulta em um *speedup* mais alto para matrizes de tamanho pequeno. No entanto, à medida que o tamanho das matrizes aumenta, esse ganho de desempenho diminui. Isso ocorre porque o tempo gasto na sincronização dos threads cresce com o aumento da carga de trabalho. Além disso, a eficiência tende a diminuir conforme o número de threads aumenta, devido ao aumento de acessos simultâneos à memória principal, o que cria um *bottleneck,* especialmente em matrizes maiores.

Nos gráficos do algoritmo V2, é possível observar que os dados estão incompletos. Isso acontece porque a execução do algoritmo é tão demorada que não foi viável realizar comparações com mais de 8 threads. Mesmo com apenas 4 threads, o desempenho já é significativamente inferior ao do algoritmo V1, sendo ainda pior com 8 threads. Este baixo desempenho deve-se à necessidade de sincronização dos threads ao final de cada iteração do loop interno, o que gera um atraso a cada linha processada. Quanto maior o número de threads, maior o tempo gasto com sincronização, aumentando ainda mais esse atraso.

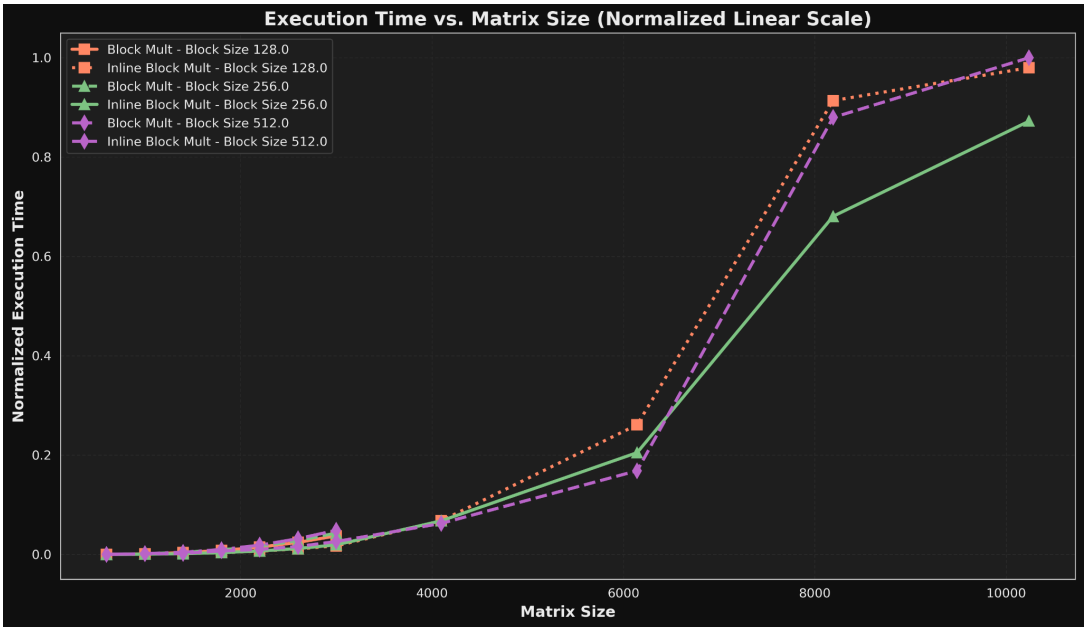
Conclusão

Este trabalho proporcionou uma oportunidade única para estudar o impacto da otimização da cache e da paralelização no desempenho de um algoritmo, permitindo-nos perceber como esses fatores podem afetar a eficiência e a escalabilidade das operações. Durante a execução, adquirimos um conhecimento mais aprofundado sobre OpenMP, uma ferramenta fundamental para a paralelização de código, e fomos capazes de explorar outras linguagens de programação, como Java e Go, para comparar diferentes abordagens na implementação dos algoritmos sugeridos. Essa experiência não só nos permitiu entender os desafios envolvidos, mas também a importância de uma implementação eficiente para maximizar o desempenho em diferentes contextos computacionais.

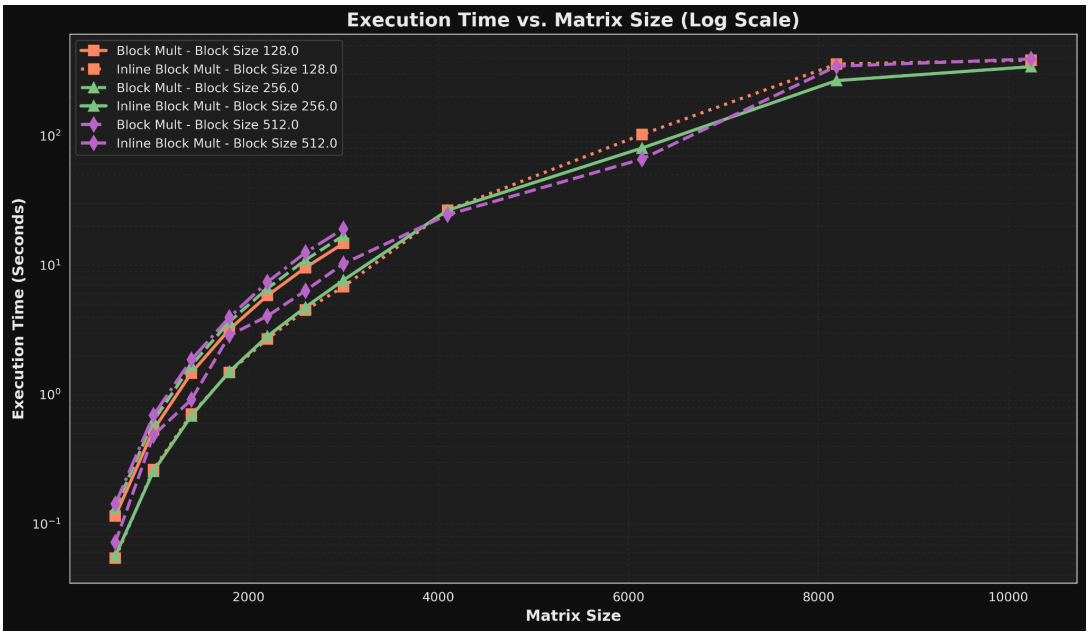
Anexos

A.1 Dados Gráficos

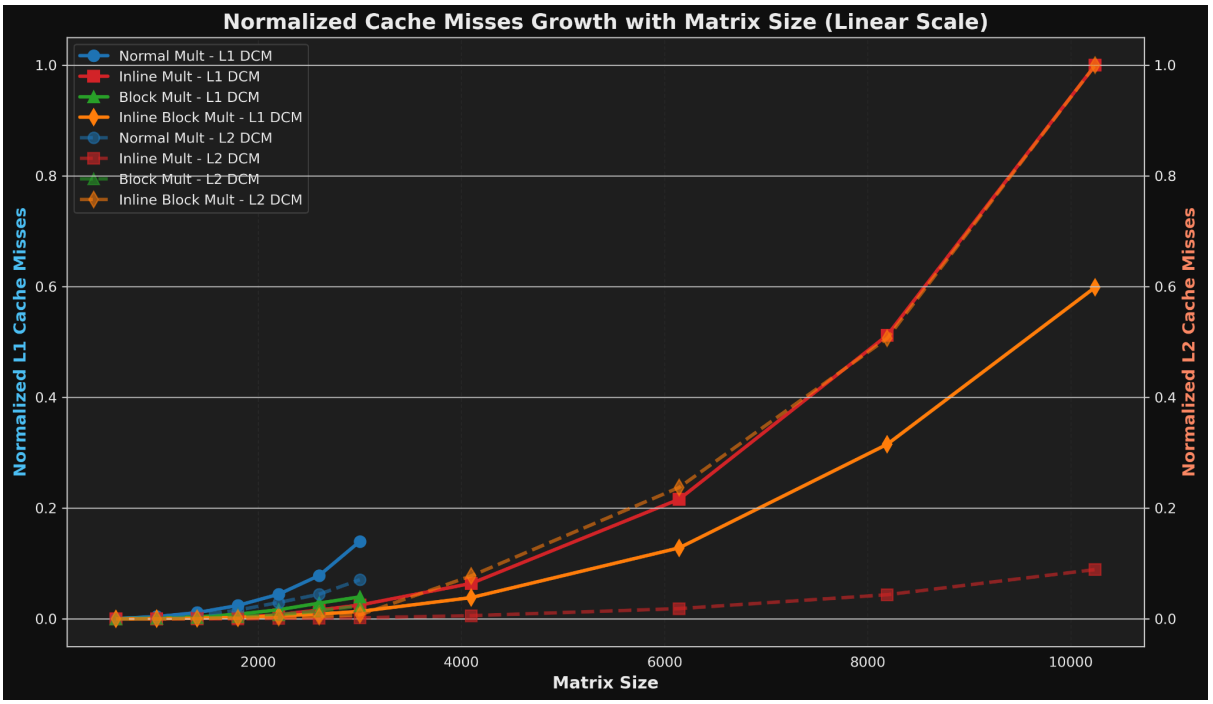
A.1.1 Tempo de execução dos algoritmos de bloco com tamanhos diferentes (C++)



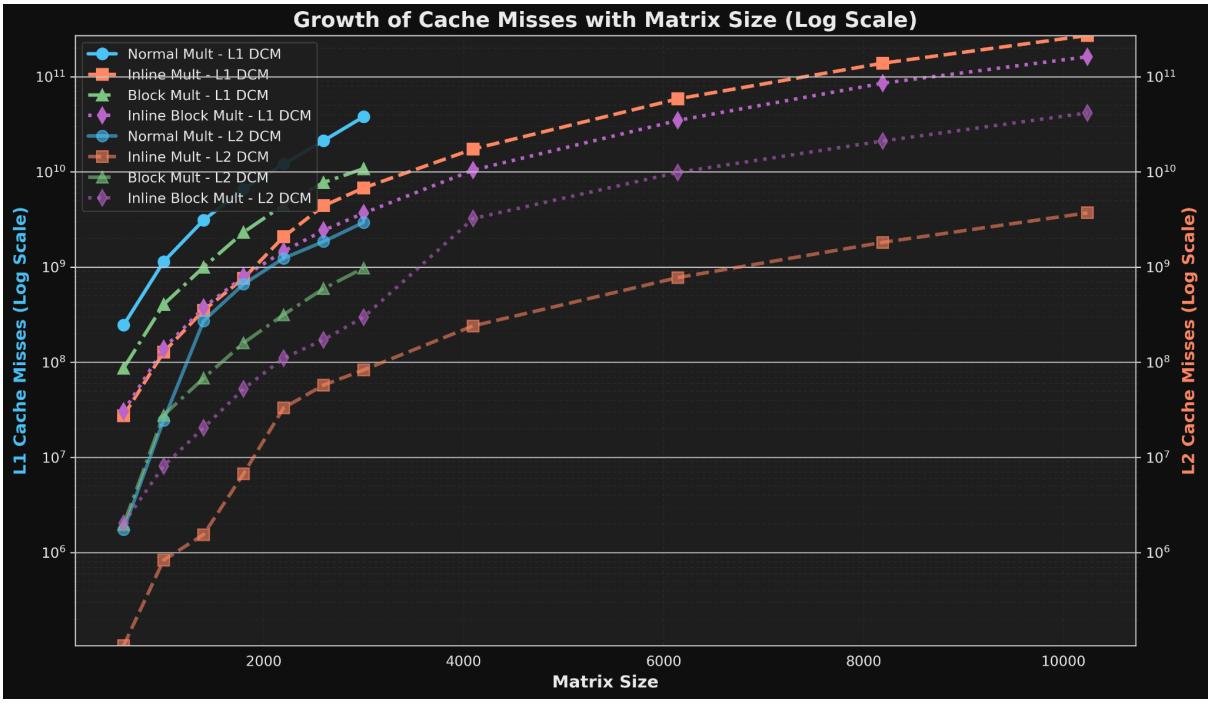
A.1.2 Tempo de execução dos algoritmos de bloco com tamanhos diferentes (C++, escala logarítmica)



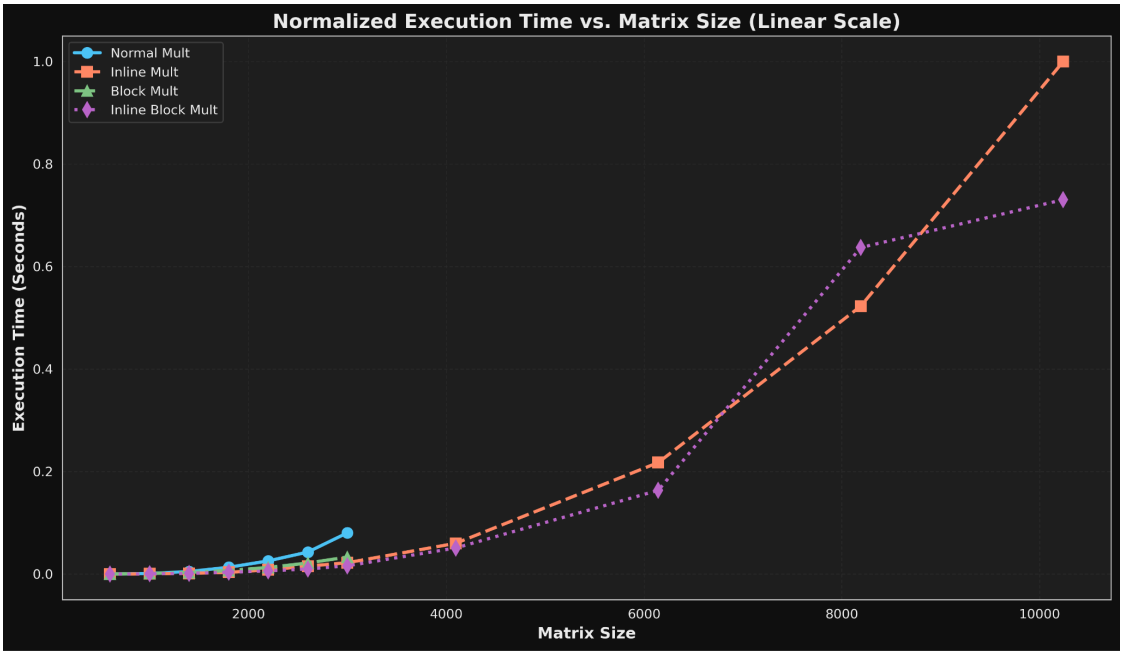
A.1.3 Número de cache misses por algoritmo (C++)



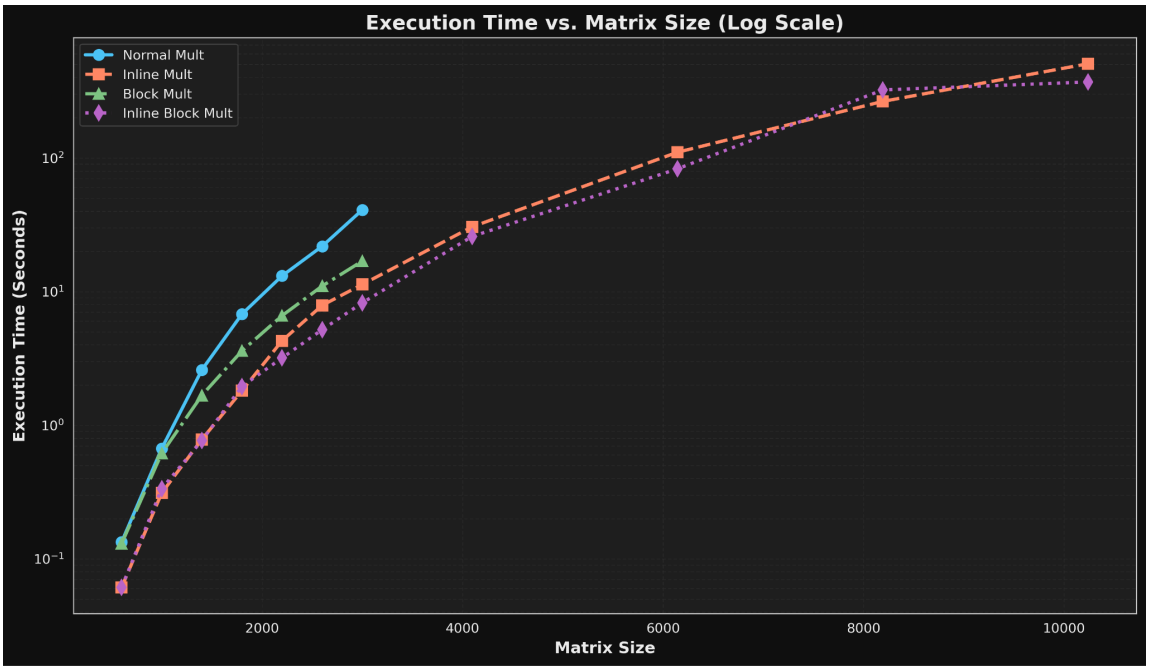
A.1.4 Número de cache misses por algoritmo (C++, escala logarítmica)



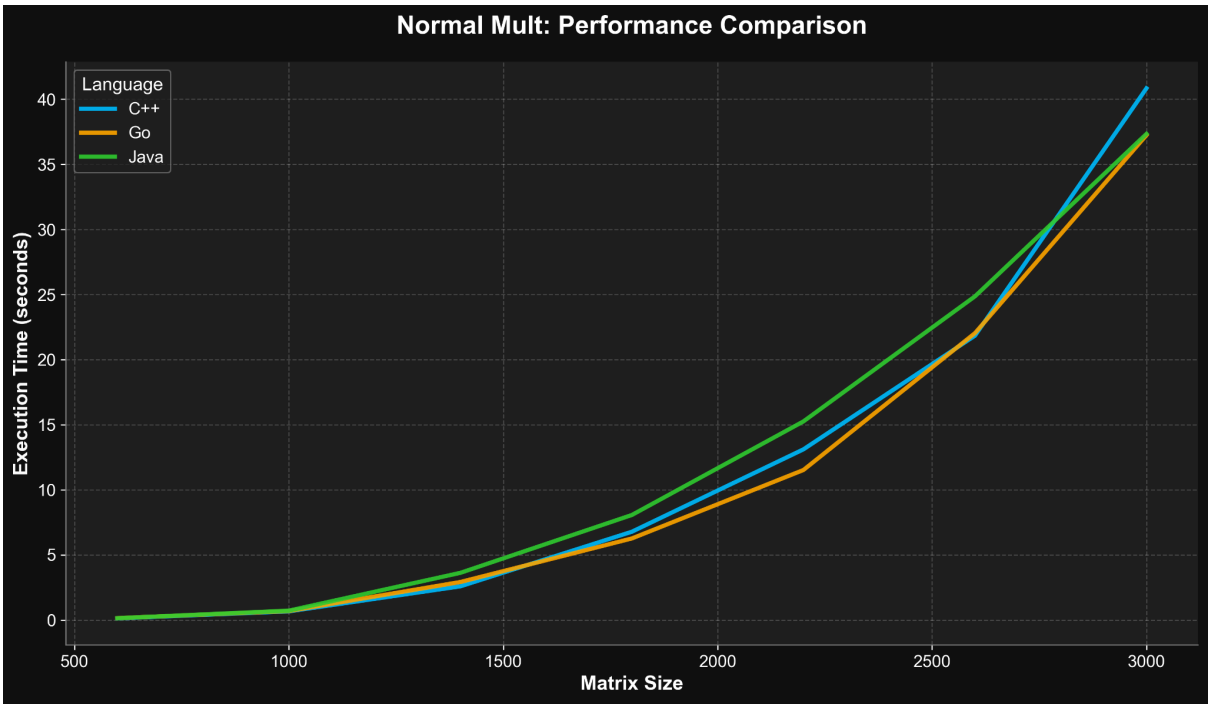
A.1.5 Tempo de execução por algoritmo (C++)



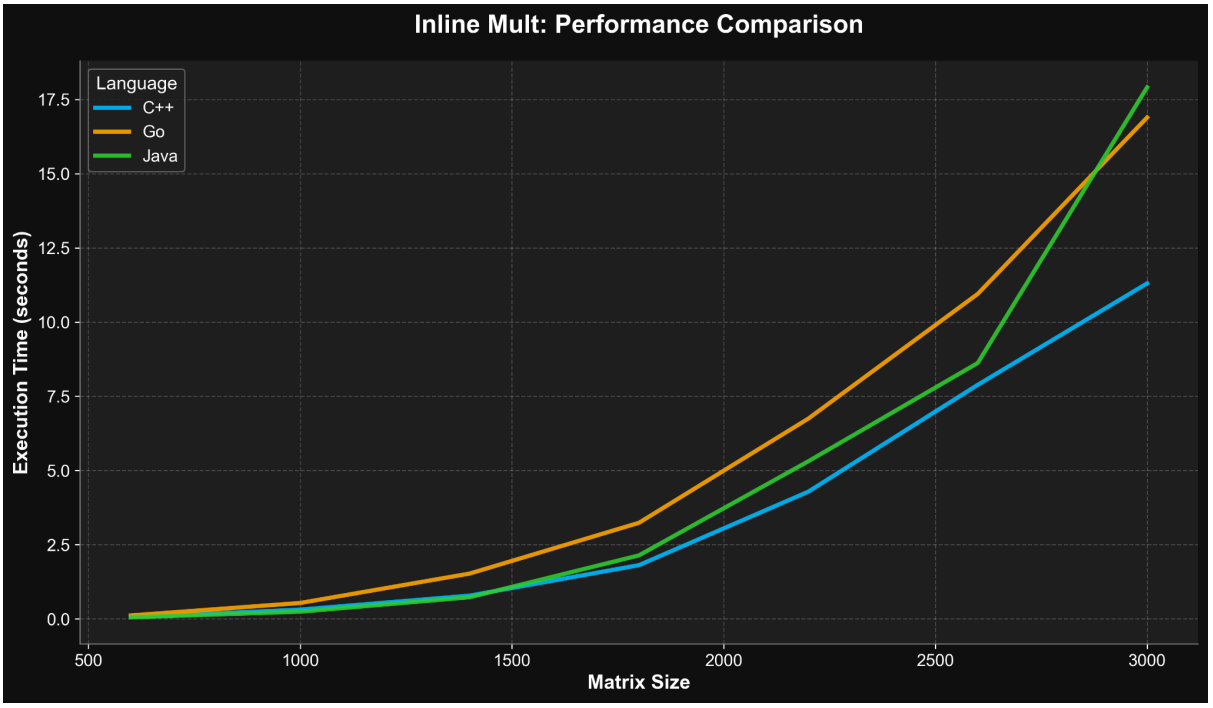
A.1.6 Tempo de execução por algoritmo (C++, escala logarítmica)



A.1.7 Tempo de execução de multiplicação convencional (C++, Java, Go)



A.1.8 Tempo de execução de multiplicação linha por linha (C++, Java, Go)



A.1.9 Comparação entre o speedup dos algoritmos de paralelização de multiplicação de matrizes (C++)

A.1.10 Comparação entre a eficiência dos algoritmos de paralelização de multiplicação de matrizes (C++)

A.1.10 Comparação de MFLOPS dos algoritmos de paralelização de multiplicação de matrizes (C++)

A.2 Pseudocódigo

A.2.1 Multiplicação Matricial Convencional

```
C/C++
for rowA from 0 to numRowsA - 1 do
    for colB from 0 to numColsB - 1 do
        sum ← 0
        for sharedDim from 0 to numColsA - 1 do
            sum ← sum + matrixA[rowA * numColsA + sharedDim] *
matrixB[sharedDim * numColsB + colB]
        end for
        matrixC[rowA * numColsB + colB] ← sum
    end for
end for
```

A.2.2 Multiplicação Matricial Linha por Linha

```
C/C++
for row from 0 to numRowsA - 1 do
    for col from 0 to numColsB - 1 do
        value ← matrixA[row * numColsA + col]
        for resultCol from 0 to numColsC - 1 do
            matrixC[row * numColsC + resultCol] ← matrixC[row * numColsC +
resultCol] + (value * matrixB[col * numColsC + resultCol])
        end for
    end for
end for
```

A.2.3 Multiplicação Matricial por Blocos

```
C/C++
for blockRowA from 0 to numRowsA - 1 step blockSize do
    for blockColB from 0 to numColsB - 1 step blockSize do
        for blockShared from 0 to numColsA - 1 step blockSize do

            minRowA ← min(blockRowA + blockSize, numRowsA)
            minColB ← min(blockColB + blockSize, numColsB)
            minShared ← min(blockShared + blockSize, numColsA)

            for rowA from blockRowA to minRowA - 1 do
                for colB from blockColB to minColB - 1 do
                    sum ← 0
                    for sharedDim from blockShared to minShared - 1 do
                        sum ← sum + matrixA[rowA * numColsA + sharedDim] *
matrixB[sharedDim * numColsB + colB]
                    end for
                    matrixC[rowA * numColsB + colB] ← matrixC[rowA *
numColsB + colB] + sum
                end for
            end for
        end for
    end for
end for
```

A.2.4 Multiplicação Matricial por Blocos Linha por Linha

```
C/C++
for blockRowA from 0 to numRowsA - 1 step blockSize do
    for blockColB from 0 to numColsB - 1 step blockSize do
        for blockShared from 0 to numColsA - 1 step blockSize do

            minRowA ← min(blockRowA + blockSize, numRowsA)
            minColB ← min(blockColB + blockSize, numColsB)
            minShared ← min(blockShared + blockSize, numColsA)

            for rowA from blockRowA to minRowA - 1 do
                for colB from blockColB to minColB - 1 do
                    valueA ← matrixA[colB + rowA * numColsB]
                    for sharedDim from blockShared to minShared - 1 do
                        matrixC[sharedDim + rowA * numColsB] ←
matrixC[sharedDim + rowA * numColsB] +
                                                                (valueA *
matrixB[sharedDim + colB * numColsB])
                    end for
                end for
            end for

        end for
    end for
end for
```

A.2.5 Multiplicação Matricial Convencional com paralelização V1

```
C/C++
#pragma omp parallel for
for rowA from 0 to numRowsA - 1 do
    for colB from 0 to numColsB - 1 do
        sum ← 0
        for sharedDim from 0 to numColsA - 1 do
            sum ← sum + matrixA[rowA * numColsA + sharedDim] *
matrixB[sharedDim * numColsB + colB]
        end for
        matrixC[rowA * numColsB + colB] ← sum
    end for
end for
```

A.2.6 Multiplicação Matricial Convencional com paralelização V2

```
C/C++
#pragma omp parallel private(rowA, colB, sum)
for rowA from 0 to numRowsA - 1 do
    for colB from 0 to numColsB - 1 do
        sum ← 0
        #pragma omp parallel for reduction(+:sum)
        for sharedDim from 0 to numColsA - 1 do
            sum ← sum + matrixA[rowA * numColsA + sharedDim] *
matrixB[sharedDim * numColsB + colB]
        end for
        matrixC[rowA * numColsB + colB] ← sum
    end for
end for
```

A.2.7 Multiplicação Matricial Linha por Linha com paralelização V1

```
C/C++
#pragma omp parallel
{
    #pragma omp for private(colB, sharedDim)
    for rowA from 0 to numRowsA - 1 do
        for colB from 0 to numColsB - 1 do
            valueA ← matrixA[rowA * numColsA + colB]
            for sharedDim from 0 to numColsB - 1 do
                matrixC[rowA * numColsB + sharedDim] ← matrixC[rowA *
numColsB + sharedDim] +
                                                        (valueA *
matrixB[colB * numColsB + sharedDim])
            end for
        end for
    end for
}
```

A.2.8 Multiplicação Matricial Linha por Linha com paralelização V2

```
C/C++
#pragma omp parallel
{
    for rowA from 0 to numRowsA - 1 do
        for colB from 0 to numColsB - 1 do
            valueA ← matrixA[rowA * numColsA + colB]

            #pragma omp for
            for sharedDim from 0 to numColsB - 1 do
                matrixC[rowA * numColsB + sharedDim] ← matrixC[rowA *
numColsB + sharedDim] +
                                                        (valueA *
matrixB[colB * numColsB + sharedDim])
            end for
        end for
    end for
}
```