Faculdade de Engenharia da Universidade do Porto

CPD Project 1

Turma 10 - Grupo 17

João Macedo Lima up202108891
Ntsay Zacarias up202008863
Rodrigo Rodrigues up202108749

17 de Março de 2023

Contents

1 Problem Description and Algorithms

O objetivo principal deste projeto foi estudar o efeito da performance do processador quando este tem que aceder a grandes quantidades de "data". Para tal, foram utilizados algoritmos de produto matricial e o PAPI (Performance API) para podermos recolher relevantes indicadores de performance (durante a execução dos algoritmos).

Relativamente à segunda parte do trabaho, foram implementadas duas versões paralelas do segundo algortimo e observamos a performance destas mesmas implementações.

1.1 Simple Matrix Multiplication

Este algoritmo correspondeu ao primeiro algoritmo do trabalho. Inicialmente foi-nos dada a implementação em C++ e o objetivo foi implementar este mesmo algoritmo noutra linguagem, neste caso, utilizamos java. Este programa multiplica uma linha da primeira matriz por cada coluna da segunda matriz. Pseudo-code in Java:

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

1.2 Line Matrix Multiplication

O "Line Matrix Multiplication" foi o segundo algoritmo a ser implementado no nosso projeto. Esta versão é uma melhor implementação do primeiro algoritmo porque em vez de multiplicar uma linha da primeira matrix pela respetiva coluna da segunda matriz, o programa acaba por multiplicar um elemento da primeira matrixz pela linha respetiva da segunda matriz. Apesar de ser mais rápido, acaba por ter a mesma complexidade $\mathbf{O(n^3)}$. Pseudo-code in java:

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

1.3 Block Matrix Multiplication

O "Block Matrix Multiplication" foi o terceiro algoritmo a ser impleemntado e corresponde a uma nova versão do segundo algoritmo onde este divide as matrizes em blo-

cos(matrize)s mais pequenos, faz o calculo usando o mesmo raciocínio do segundo algoritmo e acaba por adicionar tudo no fim.

Esta implementação deverá melhorar o tempo de acesso à "data" que está instalada na memória permitindo que haja mais "data" a ser guardada mais frequentemente em níveis de memória mais baixos e mais rápidos. Isto fará com que haja uma redução do tempo de extração de "data" a partir dos niveis superiores de memória. Pseudo-code in C/C++ :

```
for(int ii = 0; ii < m_ar; ii += bkSize) {
 for(int jj = 0; jj < m_br; jj += bkSize) {
  for(int kk = 0; kk < m_ar; kk += bkSize) {
   for(int i = ii; i < std::min(ii + bkSize, m_ar); i++) {
    for(int k = kk; k < std::min(kk + bkSize, m_ar); k++) {
     double tmp = pha[i*m_ar + k];
      for(int j = jj; j < std::min(jj + bkSize, m_br); j++) {
       phc[i*m_br + j] += tmp * phb[k*m_br + j];
      }
    }
   }
  }
 }
}
```

2 Performance Metrics

De forma a avaliar a performance dos nossos algoritmos, usamos a "Performance API (PAPI)", que acabou por nos dar acesso a várias informações acerca do CPU, tais como: o número de "Floating Points Operations" (FLOP), o número de falhas de cache tanto apra L1 como para L2, o número total de ciclos e o número total de instruções conclúidas. O número total de ciclos e de instruções completas conseguem dar uma melhor perspetiva sobre a eficiência do nosso programa porque, quanto menores forem estes números, mais rápido e eficiente será o nosso programa. Para além disso, uma operação de falha de cache implica uma considerável sobrecarga no processamento, portanto, a variação deste valor é extremamente relevante para avaliar a eficiência da nossa implementação. Para realizar os nossos testes, compilamos o nosso programa usando a **-O2** "flag" e foi utilizado um dos computadores da FEUP.

3 Results and Analysis

Os resultados são a média de duas medições consecutivas e decidimos apresentá-las através do uso de gráficos.

3.1 Tempo de execução e perdas de cache nas diferentes implementações dos algoritmos

Nestes algoritmos podemos observar uma grande diferença no tempo de execução entre os três algoritmos, sendo o mais rápido o algoritmo em bloco. Para além disso, as observamos que as implementações paralelas são mais rápidas do que a implementação original do

segundo algoritmo. onseguimos também ver que as implementações em C/C++ foram mais rápidas e eficientes do que as versões em java.

Relativamente às perdas, conseguimos observar que as perdas diminuem do primeiro para o segundo algoritmos e do segundo para o terceiro algoritmo. A primeira observação deve-se ao facto de no segundo algoritmo usarmos a matriz resulatnte como acumulador e usando também o ciclo intermédio para aceder à matrix linha por linha, o que acaba por garantir uma maior disponibilidade de "data" num nível de memória menor. Quanto à segunda observação, a diminuição de perdas verfica-se no nível L1 mas não se verifica no nível L2. Para além isso, verificamos que o tamanho do bloco influenciou nas perdas de cache onde, quanto maior fosse o tamanho do bloco, menores seríam mais eficientes.

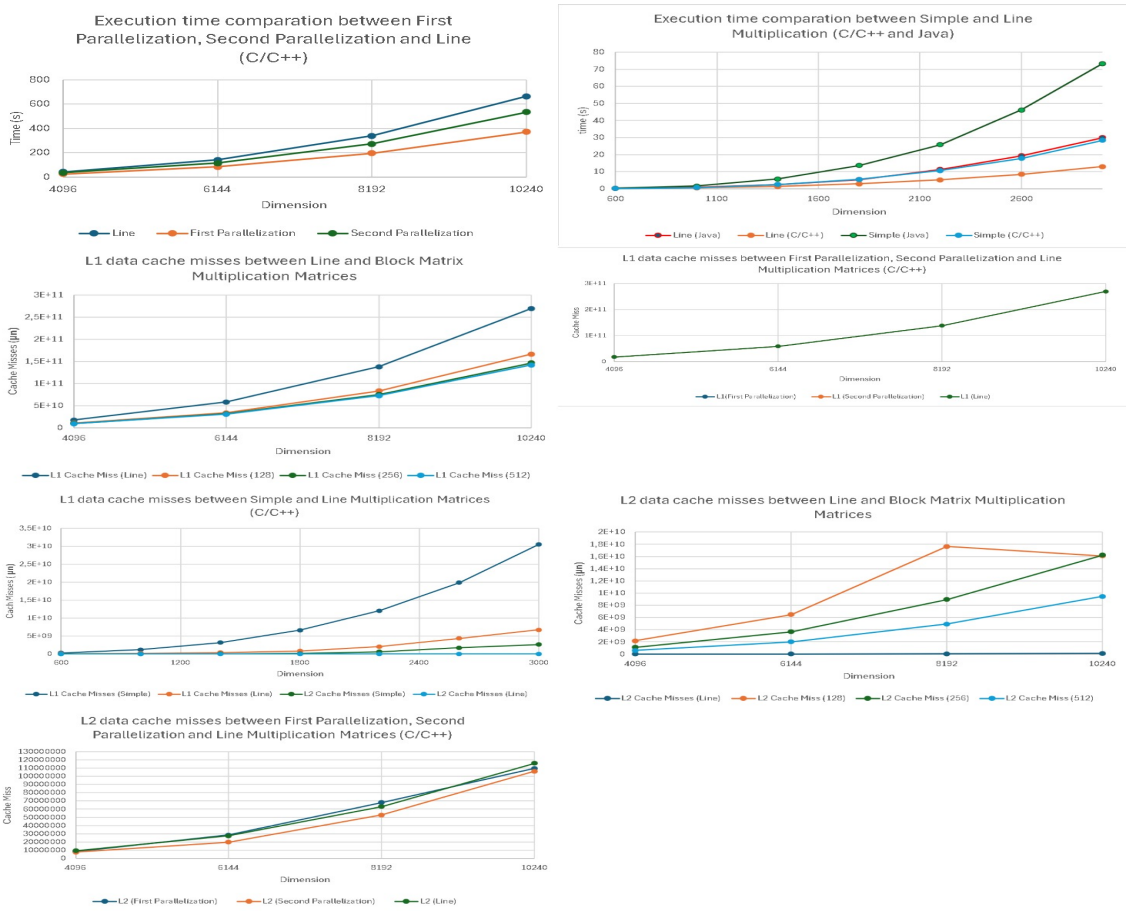

Figure 1

3.2 MFLOPS Comparison between Simple and Line Matrix Multiplication Algorithms

Como era esperado, o número de FLOP foi mantido entre entre os algoritmos visto que o número de operações foi permanecendo mais ou menos constante. Para além disso, o alto número de FLOP por segundo no segundo algoritmo está relacionado com as baixas perdas de cache nos níveis L1 e L2. Para concluir, conseguimo tamém observar que o primeiro algoritmo tende a ser mais lento à medida que a dimensão vai crescendo.

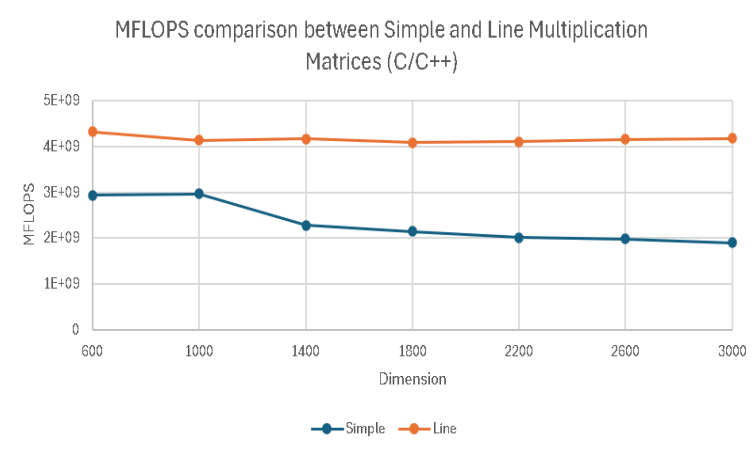

Figure 2

3.3 MFLOPS Comparison between Line and Block Matrix Multiplication Algorithms

Como seria esperado, os FLOP mantiveram-se entre os algoritmos visto que o número de operações se manteve mais ou menos constante (como pudemos observar a última comparação).

Para além disso, será de realçar que existe uma ligeira diferença com o aumento do tamanho do bloco. Isto deve-se ao facto de o bloco 128x128 não usar inteiramente nível L2 da cache para as suas FLOP.

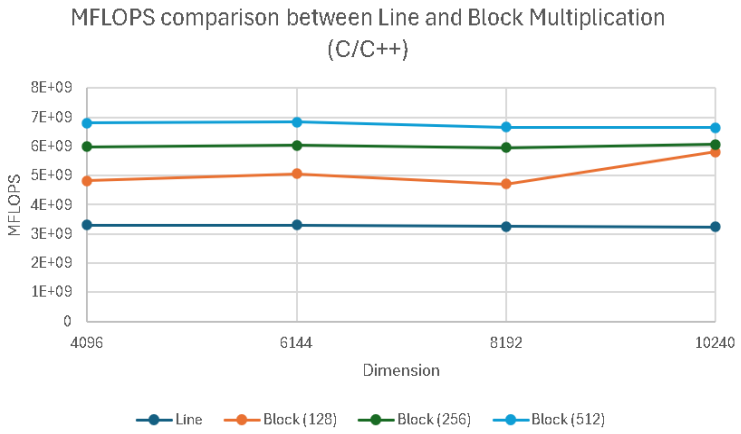

Figure 3

3.4 MFLOPS, SpeedUp and efficiency of the Parallel Implementations

Novamente como seria esperado, foi possível observar que os FLOP mantiveram-se, visto que o número de operações se manteve constante. Será também de realçar que a primeira paralelização é mais eficente porque podemos observar que o número de MFLOPS é superior reativamente à segunda paralelização. Para além disso, o "Speed Up" e a eficiência são

superiores na primeira paralelização, oq ue acaba por ser coerente com o que observamos anteriormente (na comparação dos MFLOPS entre as duas implementações).

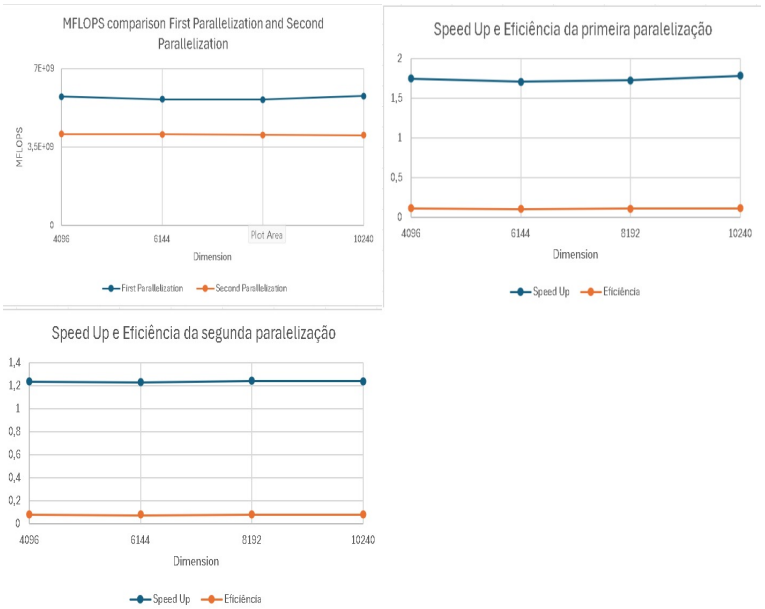

Figure 4

4 Conclusion

Este projeto demonstrou a importância de estratégias eficientes de acesso à memória e paralelismo na maximização do desempenho em operações computacionais intensivas. A análise revelou a multiplicação simples como sendo a menos eficaz, enquanto que a abordagem por linha e, mais notavelmente, a multiplicação por blocos, que usam formas semelhantes de acesso à memória, melhoraram significativamente tanto a eficiência da cache como o tempo de execução. Porém, estas implementações fizeram uso de apenas um core. Adicionalmente, as implementações paralelas realçaram o impacto positivo do paralelismo na aceleração destas operações. Assim, ficou evidente para nós, que estas técnicas são cruciais para otimizar o desempenho de tarefas de alta demanda computacional.