Performance Evaluation of a single and multi-core

Este projeto teve como objetivo o **estudo da performance do processador na hierarquia de memória** ao aceder tamanhos de dados altos, usando o produto de duas matrizes como objeto de estudo.

Isso será feito através de duas partes distintas: na primeira parte, será analisado o impacto da hierarquia de memória em um único núcleo de processamento, enquanto na segunda parte, serão investigadas implementações paralelas em sistemas multi-core.

A Performance API (PAPI) será utilizada para coletar indicadores de desempenho relevantes.

Grupo: Gabriel Machado Jr Guilherme Araujo Sofia Valadares

Performance Evaluation of a single core

Na primeira parte do projeto, foi requisitado a implementação de diferentes versões do algoritmo de produto de duas matrizes. De modo a facilitar a leitura do relatório.
Nesta parte do projeto, será realizada uma avaliação de desempenho de um único núcleo de processamento. O foco será entender como a hierarquia de memória impacta o desempenho ao acessar grandes volumes de dados, utilizando a multiplicação de matrizes como exemplo. Foram implementadas 3 diferentes versões de um algoritmos do produto de duas matrizes, ultilizando de duas diferentes liguagens, C++ e Python.

1. MULTIPLICAÇÃO LINHA X COLUNA

Nesta abordagem, o algoritmo implementado multiplica uma linha da primeira matriz por cada coluna da segunda matriz. Esse algoritimo foi implenetado em C++ e Pyhton.

Nesta implementação, temos três loops aninhados. O loop externo percorre as linhas da matriz resultante `phc`. Para cada linha, o segundo loop percorre as colunas da matriz resultante `phc`. Dentro desses loops, outro loop é utilizado para calcular cada elemento da matriz resultante, que é a soma dos produtos dos elementos correspondentes das linhas da matriz `pha` e das colunas da matriz `phb`. A implementaçao nas duas linguagens segue uma mesma logica, o que pode mudar para cada uma é a sintax das liguagens.

Implementação em C++:

```
Time1 = clock();

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;
```

```
    }
}

Time2 = clock();
sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1) /
CLOCKS_PER_SEC);
cout << st;
```

Nesta implementação em C++, temos três loops aninhados. O loop externo percorre as linhas da matriz resultante `phc`. Para cada linha, o segundo loop percorre as colunas da matriz resultante `phc`. Dentro desses loops, outro loop é utilizado para calcular cada elemento da matriz resultante, que é a soma dos produtos dos elementos correspondentes das linhas da matriz `pha` e das colunas da matriz `phb`.

Implementação em Python:

```
Time1 = time.time()

for i in range(m_ar):
    for j in range(m_br):
        temp = 0
        for k in range(m_ar):
            temp += pha[i*m_ar + k] * phb[k*m_br + j]
        phc[i*m_ar + j] = temp

Time2 = time.time()
print(f"Time: {((Time2 - Time1)):.3f} seconds\n")
```

A implementação em Python segue uma lógica semelhante à implementação em C++, mas usando a sintaxe e estruturas de controle específicas do Python. Aqui, também temos três loops aninhados. O primeiro loop `for i` percorre as linhas da matriz resultante `phc`, o segundo loop `for j` percorre as colunas da matriz resultante `phc`, e o terceiro loop `for k` calcula cada elemento da matriz resultante, semelhante à lógica da implementação em C++.

O tempo de processamento é registrado para matrizes de entrada variando de 600x600 a 3000x3000 elementos, com incrementos de 400 em ambas as dimensões, em ambas as linguagens. No compilador C++, a flag de otimização -O2 é utilizada para garantir uma compilação otimizada.

Tamanho da Matriz	C++ (segundos)	Python (segundos)
600	0.192	32.082
1000	1.194	158.198
1400	3.369	439.760
1800	17.918	946.323
2200	39.745	1731.402
2600	70.490	x

Tamanho da Matriz	C++ (segundos)	Python (segundos)
3000	118.942	3502.117 *1

Observação *1: Esse texte foi feito em uma maquina diferente devido a indisponibilidade de tempo e maquinas no momento do trabalho.

Observa-se que o tempo de processamento aumenta conforme o tamanho da matriz, quando em valores maiores, mesmo que esse almento tenha o mesmo valor, o almento de tempo é maior. Além disso, é notável que a implementação em C++ é significativamente mais rápida do que a implementação em Python, e essa diferença de desempenho aumenta à medida que o tamanho da matriz aumenta. Essas análises fornecem insights valiosos sobre o desempenho relativo das duas linguagens na execução do mesmo algoritmo de multiplicação de matrizes.

2. MULTIPLICAÇÃO LINHA X LINHA

Nesta versão, realizamos a multiplicação de cada linha da primeira matriz por cada linha da segunda matriz. Isso significa que, em vez de multiplicar uma linha da primeira matriz por cada coluna da segunda matriz, como na versão anterior, multiplicamos cada linha da primeira matriz por cada linha da segunda matriz. Essa abordagem resulta em uma diferente estrutura de laços no código.

Nesta implementação, são usados três loops aninhados. O primeiro loop `for(i)` itera sobre as linhas da matriz resultante `phc` e também representa as linhas da matriz 'pha'. Dentro deste loop, o segundo loop `for(j)` passa pelas linhas da matriz `phb`. Por fim, o terceiro loop `for(k)` itera sobre as linhas da matriz resultante `phc` e é utilizado para calcular cada elemento da matriz resultante `phc`. Dentro deste último loop, cada elemento `phc[i*m_ar + k]` é incrementado pelo produto dos elementos correspondentes das linhas da matriz `pha` e das linhas da matriz `phb`.

Implementação em C++:

```
Time1 = clock();

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

Time2 = clock();
sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1) /
CLOCKS_PER_SEC);
cout << st;
```

Nesta implementação em C++, são usados três loops aninhados. O primeiro loop `for(i)` itera sobre as linhas da matriz resultante `phc` e também representa as linhas da matriz 'pha'. Dentro deste loop, o segundo loop `for(j)` passa pelas linhas da matriz `phb`. Por fim, o terceiro loop `for(k)` itera sobre as linhas da matriz resultante `phc` e é utilizado para calcular cada elemento da matriz resultante `phc`. Dentro deste último loop, cada elemento `phc[i*m_ar + k]` é incrementado pelo produto dos elementos correspondentes das linhas da matriz `pha` e das linhas da matriz `phb`.

Implementação em Python:

```
Time1 = time.time()

for i in range(m_ar):
    for j in range(m_ar):
        for k in range(m_br):
            phc[i*m_ar + k] += pha[i*m_ar + k] * phb[j*m_ar + k]

Time2 = time.time()
print(f"Time: {((Time2 - Time1)):.3f} seconds\n")
```

A implementação em Python segue uma lógica semelhante à implementação em C++, usando loops `for` aninhados para percorrer as linhas e colunas da matriz resultante `phc`. O loop `for i in range(m_ar)` percorre as linhas da matriz resultante, o loop `for j in range(m_ar)` percorre as colunas, e o loop `for k in range(m_br)` é utilizado para calcular cada elemento da matriz resultante. Dentro deste último loop, cada elemento `phc[i*m_ar + k]` é incrementado pelo produto dos elementos correspondentes das linhas da matriz `pha` e das colunas da matriz `phb`.

Assim, foram realizados os testes de desempenho para ambas as implementações.

Tamanho da Matriz	C++ (segundos)	Python (segundos)
600	0.103	43.061
1000	0.492	199.246
1400	1.637	549.308
1800	3.714	1148.476
2200	6.972	2119.244
2600	12.087	4070.254
3000	17.878	3309.937 *1

Observação *1: Este teste foi realizado em uma máquina diferente devido à indisponibilidade de tempo e máquinas no momento do trabalho.

Foi observado que o tempo de processamento aumenta conforme o tamanho da matriz. Além disso, assim como na versão anterior, a implementação em C++ apresenta um desempenho significativamente melhor em relação à implementação em Python, com a diferença de desempenho tornando-se mais pronunciada à medida que o tamanho da matriz aumenta

Foi obserevadp tudo aquilo que foi observado na versão anterior (1. MULTIPLICAÇÃO LINHA X COLUNA), mais também com a adiçao que o algoritimo em C++ se mostrou mais rapido e com menor almento de tempo conforme a matriz almentava, ao contrario dos testes em Python que se mostraram mais lentos em coparaçao ao primeiro algotitimo (a exeçao do teste 3000 que ambos foram relalizados em outra maquina e nesse caso o segundo algoritimo se provou ligeiramente mais rapido).

Também foram realizados testes apenas em C para tamanhos de matriz maiores:

Tamanho da Matriz	C (segundos)
4096	48.913
6144	164.480
8192	411.651
10240	792.512

Oque agente observa aqui?????: Nesses testes podemos ver a real eficiencia do segundo algoritimo em C++ em relaçao ao primeiro, devido aos primeiros teste se mostrar com menor tempo que aos dois ultimos testes do primeiro algoritimo, e também em relação ao de Python, já que mesmo com numeros maiores o tempo de processamento se mostra inferior a varios testes realizados no algoritimo de python

3. MULTIPLICAÇÃO POR BLOCO

Nesta abordagem, utilizamos o conceito de multiplicação por blocos para otimizar o desempenho da multiplicação de matrizes. Essa técnica envolve dividir as matrizes em blocos menores e realizar a multiplicação de cada bloco individualmente. Isso é feito para reduzir o número de acessos à memória e maximizar a eficiência do cache.

```
    Time1 = clock();

    for (ib=0; ib<m_ar; ib+= bkSize)
    {
        for (jb=0; jb<m_br; jb+=bkSize)
        {
            for (kb=0; kb<m_ar; kb+=bkSize)
            {
                for(i=ib; i<min(ib+bkSize, m_ar); i++)
                {
                    for(j=jb; j<min(jb+bkSize, m_br); j++)
                    {
                        for(k=kb; k<min(kb+bkSize, m_ar); k++)
                        {
                            phc[i*m_ar+j] += pha[i*m_ar+k] *
phb[k*m_br+j];
                        }
                    }
                }
            }
        }
```

```
    }

    Time2 = clock();
    sprintf(st, "Time: %3.3f seconds\n", (double)(Time2 - Time1) /
CLOCKS_PER_SEC);
    cout << st;
```

O algoritmo implementado em C++ funciona da seguinte maneira:

1. O loop externo `for (ib=0; ib<m_ar; ib+= bkSize)` percorre as linhas da matriz resultante `phc` em blocos de tamanho `bkSize` na dimensão das linhas.
2. Dentro deste loop, há outro loop `for (jb=0; jb<m_br; jb+=bkSize)` que percorre as colunas da matriz resultante `phc` em blocos de tamanho `bkSize` na dimensão das colunas.
3. Em seguida, há um terceiro loop `for (kb=0; kb<m_ar; kb+=bkSize)` que percorre as linhas da matriz `pha` em blocos de tamanho `bkSize` na dimensão das linhas.
4. Dentro desses três loops, há quatro loops aninhados para calcular os elementos de cada bloco da matriz resultante `phc`. Esses loops percorrem as linhas (`i`), as colunas (`j`) e as linhas da matriz `pha` (`k`) dentro dos blocos, limitados pelo tamanho do bloco e as dimensões das matrizes.
5. Dentro do loop mais interno, o elemento `phc[i*m_ar+j]` é calculado como a soma acumulada dos produtos dos elementos correspondentes das linhas da matriz `pha` e das colunas da matriz `phb`.

Essa abordagem permite reduzir os acessos à memória e otimizar o uso do cache, resultando em um desempenho melhorado em comparação com os métodos tradicionais de multiplicação de matrizes.

Os testes foram realizados variando o tamanho do bloco (`bkSize`) e o tamanho da matriz, com os seguintes resultados:

Tamanho da Matriz	Tempo bkSize = 128 (segundos)	Tempo bkSize = 256 (segundos)	Tempo bkSize = 512 (segundos)
4096	89.187	91.182	375.748
6144	305.734	300.610	314.221
8192	723.451	3817.102	x
10240	1422.087	1407.182	x

Esses resultados demonstram que o desempenho varia significativamente com o tamanho do bloco e do tamanho da matriz. Observa-se um aumento no tempo de execução à medida que o tamanho do bloco aumenta para casos de valores menores, já em valores menores se observa geralmente uma leve queda de tempo execução com blocos menores (O pq disso sla tem que ver ai).

Ainda sim se formos comparar os resoltados com o segundo algoritimo em relaçao a tempo de execução a multiplicaçao em bloco se mostra menos eficiente. Porem ela poderia ser sim mais eficiete em relaçao ao primeiro algoritimo, para chegarmos a conclusoes mais concretas deveriam ser realizados testes como os valores de matriz nesse traste ultitizados no primeiro algoritimo

Em relação aos testes realizados apenas em C, os tempos de execução variam dependendo do tamanho do bloco e da matriz. Esses resultados podem fornecer insights úteis para a seleção dos parâmetros ideais de

tamanho de bloco para otimização do desempenho do algoritmo de multiplicação de matrizes. Realmente precisa dessa parte????

Performance evaluation of a multi-core implementation

Na segunda parte deste projeto, será realizada uma avaliação de desempenho de implementações multi-core. Serão implementadas versões paralelas da multiplicação de matrizes, especificamente a implementação linha a linha. Duas soluções paralelas distintas serão propostas e analisadas em termos de MFlops, aceleração (speedup) e eficiência. O objetivo é comparar o desempenho das soluções paralelas em sistemas multi-core e avaliar sua eficácia em relação à implementação sequencial.

A diferença entre os dois algoritmos está na forma como as iterações são paralelizadas:

Algoritmo 1:

```
Time1 = clock();
#pragma omp parallel for
for(i=0; i<m_ar; i++)
{
    for(j=0; j<m_ar; j++)
    {
        for(k=0; k<m_br; k++)
        {
            phc[i*m_ar+k] += pha[i*m_ar+k] * phb[j*m_ar+k];
        }
    }
}
Time2 = clock();
```

Neste algoritmo, a diretiva `#pragma omp parallel for` é utilizada para paralelizar o loop externo `for(i=0; i<m_ar; i++)`. Isso significa que cada iteração desse loop é distribuída entre várias threads para serem executadas em paralelo. No entanto, as iterações internas dos loops `j` e `k` ainda são executadas sequencialmente dentro de cada thread.

Algoritmo 2:

```
Time1 = clock();
#pragma omp parallel
for(i=0; i<m_ar; i++)
{
    for(j=0; j<m_ar; j++)
    {
        #pragma omp for
        for(k=0; k<m_br; k++)
        {
            phc[i*m_ar+k] += pha[i*m_ar+k] * phb[j*m_ar+k];
        }
    }
```

```
    }
    Time2 = clock();
```

Neste segundo algoritmo, a diretiva `#pragma omp parallel` é usada para criar um grupo de threads, e a diretiva `#pragma omp for` é aplicada no loop interno `for(k=0; k<m_br; k++)`. Isso significa que cada iteração do loop interno é distribuída entre as threads disponíveis no grupo, permitindo que diferentes threads processem diferentes partes do cálculo em paralelo.

Em resumo, enquanto o primeiro algoritmo paraleliza apenas o loop externo, distribuindo as iterações entre threads separadas, o segundo algoritmo paraleliza também o loop interno, permitindo que partes específicas do cálculo sejam executadas simultaneamente em diferentes threads. A escolha entre os dois depende das características específicas do problema e do hardware disponível.

Fazer testess e achar coisas blaus. Fazemos em no mac sem a api e colocamos uma obs para o professor