Avaliação de performance

Relatório 1º Projeto

Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela

4º ano – 2º semestre

Trabalho efetuado por:

- Cláudia Margarida da Rocha Marinho – up201404493@fe.up.pt

9 de março de 2018

1. Descrição do problema e explicação dos algoritmos

No âmbito da unidade curricular de Computação Pararela, foi proposta a realização de um estudo do efeito da hierarquia de memória no desempenho do processador quando se acede a uma volumosa quantidade de dados.

Para concretizar este estudo, foi necessária a implementação de um algoritmo de multiplicação de matrizes em diferentes linguagens de programação, nomeadamente C++, Java, Python e C# no meu caso. No entanto, a maior parte do estudo centrou-se na implementação em C++ já que nesta implementação foram desenvolvidas e analisadas duas versões do algoritmo de multiplicação de matrizes.

De forma a melhor avaliar a performance do processador, também foram recolhidos vários dados do mesmo durante a execução dos programas desenvolvidos com matrizes de tamanhos diferentes (nomeadamente com matrizes de tamanho 500x500, 1000x1000, etc.).

Neste relatório, vai-se então efetuar uma análise e comparação dos resultados obtidos, de forma a explicar os resultados obtidos e a tirar conclusões apropriadas. No entanto, antes de proceder a esta análise é importante explicar os dois algoritmos implementados e porque é que se obtém performances diferentes apesar de os dois algoritmos fazerem exatamente o mesmo.

1.1. Algoritmo Naive

Esta versão do algoritmo de multiplicação de matrizes é a transformação imediata para código da definição do produto entre matrizes. A definição do produto entre matrizes só se aplica nos casos em que o número de colunas da primeira matriz é igual ao número de linhas da segunda matriz. No entanto, como neste estudo só interessam matrizes quadradas (NxN), não foi necessário preocupar com as situações em que tal não acontece na implementação do algoritmo.

O resultado da multiplicação entre matrizes quadradas NxN é igualmente uma matriz quadrada NxN e é obtida através da seguinte fórmula:

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + ... + a_{in} b_{nj}$$ para cada par i e j com $1 \leq i \leq n$ e com $1 \leq j \leq n$, em que i corresponde a uma linha e j corresponde a uma coluna da matriz produto.

Ou seja, cada elemento da matriz resultante é obtido somando as multiplicações dos elementos ao longo de uma linha da matriz A pelos elementos ao longo de uma coluna da matriz B. A implementação deste algoritmo em C++ é:

```
for (int i = 0; i < dim; i++) {
        for (int j = 0; j < dim; j++) {
                for (int k = 0; k < dim; k++) {
                        c[i][j] += a[i][k] * b[k][j];
                }
        }
}
```

No fim de cada execução do loop mais interior (loop com a variável k), vai ser possível obter o elemento i,j da matriz resultante do produto das matrizes m1 e m2. No entanto, desta forma vai haver pouco aproveitamento da informação armazenada na memória cache, já que o acesso às posições do array de duas dimensões m2 não é feita sequencialmente. Ou seja, em cada iteração do loop mais interior, para aceder à posição correspondente de m2, vai ser necessário saltar de linha em linha em vez de aproveitar a informação armazenada para o array unidimensional anterior (para um dado j, acede-se m2[0][j], m2[1][j], …, m2[n][j]). Isto vai resultar num aumento no número de *cache misses*, o que por sua vez resulta num aumento do tempo de execução do algoritmo, como se vai averiguar mais à frente.

1.2. Algoritmo Optimizado

Esta versão do algoritmo de multiplicação de matrizes tem exatamente o mesmo resultado que o algoritmo anterior, mas é implementado de forma a aproveitar ao máximo a forma como a informação das matrizes é armazenada na memória cache. Para conseguir isto, só foi necessário introduzir uma pequena alteração, como é possível perceber pela seguinte implementação em C++:

```
for (int i = 0; i < dim; i++) {
        for (int k = 0; k < dim; k++) {
                for (int j = 0; j < dim; j++) {
                        c[i][j] += a[i][k] * b[k][j];
                }
        }
}
```

Como se pode verificar no excerto de código em cima, a única diferença em relação ao algoritmo *Naive* é que o índice j foi trocado pelo índice k no loop. Isto faz com que o loop mais interior percorra j em vez de percorrer k como na versão anterior e tem implicações positivas ao nível da performance. Isto é porque agora em cada iteração do ciclo, só são acedidas posições sequenciais na memória (para um dado k, acede-se m2[k][0], m2[k][1], …, m2[k][n]), o que resulta num aumento significativo da performance em relação à versão anterior.

2. Métricas da performance e metodologia de avaliação

O algoritmo de multiplicação de matrizes foi implementado em C++, Java, Python e C#. Foram efetuados vários testes ao tempo de execução nas implementações em diferentes linguagens, usando matrizes de várias dimensões. Também foram recolhidos dados de performance durante a execução dos diferentes programas através do **tiptop**.

No caso da implementação em C++, também foram efetuados testes usando a biblioteca **PAPI** (Performance Application Programming Interface). Foram efetuados vários testes às duas versões do algoritmo de multiplicação de matrizes, de forma a ser possível comparar estes dois.

No que toca às métricas de performance mais relevantes que foram recolhidas na execução do programa em C++ teve-se o tempo real de execução do programa, o número de instruções por ciclo (IPC), FLOPS (Floating-Point Operations Per Second) e o número de *data cache misses* nos níveis 1 e 2 da cache. As FLOPS tiveram de ser calculadas manualmente, pois o meu processador não suportava os eventos relacionados com FLOP provenientes do PAPI.

Após a recolha dos resultados, também se calculou rácios de tempo entre as diferentes linguagens e entre as duas versões do algoritmo, de forma a melhor comparar as diferentes implementações. Além disso, similarmente se calculou *L1 data cache misses* por instrução e *L2 data cache misses* por instrução, de forma a ser mais fácil analisar e comparar os valores obtidos para as *data cache misses*. Também se criou alguns gráficos através do **Excel** com as informações recolhidas de forma a facilitar a análise.

No caso da implementação em C++, foram aplicadas otimizações ao compilador com a flag **-O3** na geração do executável.

Por último é importante mencionar as caraterísticas do computador em que se correu os testes: Intel® Core™ i7-4510U CPU @ 2.00GHz, 2 cores, 4 threads por core, CPU 2800MHz, L1d cache 32K, L1i cache 32K, L2 cache 256K e L3 cache 4096K.

3. Resultados e análise

Através da análise dos tempos de execução obtidos, facilmente se verifica que o tempo de execução do algoritmo optimizado é sempre menor que o do algoritmo naive nos vários testes efetuados com o programa em C++, tal como esperado. Também se verifica que o rácio entre o tempos de execuçãos dos algoritmos para uma dada dimensão das matrizes (naive/optimizado) vai aumentando ao longo do tempo, sendo que para matrizes de dimensão 10000x10000 verificou-se um rácio de 6, ou seja, verificou-se que o algoritmo optimizado executou 6 vezes mais rápido em relação ao naive!!

A análise dos tempos de execução também possibilitou a comparação da performance da implementação do algoritmo naive em diferentes linguagens de programação. Para os testes efetuados, averiguou-se que os testes em C++ são os mais rápidos, sendo aproxidamente 2 vezes mais rápidos que os de Java, 8 vezes mais rápidos que os de C# e 100+ vezes rápidos que os de Python.

Em relação à implementação em C++, é preciso ter em conta que foram aplicadas otimizações ao compilador e que sem estas, a execução em C++ era ligeiramente mais lenta que a implementação em Java. Também é importante notar que numa versão anterior do C++, eu usava arrays alocados dinamicamente (alocação no *Heap*), o que resultava em testes de C++ ainda mais lentos que os de C# para arrays de maior dimensão. Na versão final, passei a usar arrays alocados na Stack e verificou-se um aumento notório na performance. Também evitei incluir *function calls* no corpo dos algoritmos de forma a não afetar negativamente a performance.

Em relação aos resultados obtidos para as *data cache misses*, verificou-se que tanto no nível 1 como no nível 2, o número de *data cache misses* é sempre vastamente superior para o algoritmo naive em relação ao optimizado, tal como era esperado. Também se verificou

algo interessante: para matrizes de dimensão igual ou superior a 5000, o algoritmo naive passa a usar o nível 2 da cache muito mais frequentemente, pois verifica-se um aumento da percentagem de L2 data cache misses por instrução de 1% (que se verificava para matrizes menores) para 15%. É de notar que para o algoritmo naive, verifica-se uma percentagem de L1 data cache misses por instrução a 15% consistentemente.

Já o algoritmo optimizado tem uma percentagem de *data cache misses* consistemente baixa, havendo apenas um ligeiro aumento de 1.8% para 3.5% de *L1 data cache misses* por instrução quando as matrizes passam a ser maiores que 5000x5000 e verificando-se aproximadamente 0.8% de *L2 data cache misses* por instrução para matrizes de várias dimensões. Isto acontece pois o espaço da memória cache está a ser melhor aproveitado pelo algoritmo como já foi explicado.

No que toca às FLOPS, verifica-se que o valor desta métrica é sempre superior na versão optimizada. Isto deve-se ao facto do algoritmo optimizado gastar menos tempo a aceder à memória, o que resulta numa maior dedicação do processador às operações de vírgula flutuante.

No que diz respeito aos resultados obtidos através do tiptop, foi possível verificar que a percentagem de misses para o algoritmo naive era aproxidamente 4 vezes maior em relação ao algoritmo optimizado. Também se verificou para testes em que se implementou o paralelismo algo interessante: quando o número de threads era 1, a percentagem de utilização do CPU andava à volta dos 100%; quando o número de threads era 2, a mesma andava à volta dos 200%; quando o número de threads era 3 esta rondava os 300%; por último, quando o número de threads era 4 ou mais, a percentagem de utilização do CPU não ultrapassava os 400%. Isto certamente deve-se ao facto do número de *cores* do CPU corresponder exatamente a quatro, o que permite paralelizar o trabalho do processador em quatro unidades de processamento diferentes.

Por último, também se averiguou que a implementação de paralelismo aumenta significativamente a performance tanto para o algoritmo naïve como para o optimizado. Para todos os testes efetuados com um número de threads a variar desde 2 até 8, isto foi observado. Em relação ao número de threads óptimo para a implementação do algoritmo, é difícil escolher com certidão devido à variedade dos resultados obtidos e de haver pouca diferença entre os testes efetuados com mais de 1 thread. No entanto, eu penso que o

número de threads que melhora a performance ao máximo (pelo menos no meu computador) é 4 já que parece ser com este número que se obtém os melhores resultados.

4. Conclusões

Concluí-se que o algoritmo optimizado é muito mais eficiente que o algoritmo naïve, devido ao facto de este aproveitar melhor o posicionamento dos dados na memória cache, o que resulta em menor tempo de execução, mais FLOPS e menos *data cache misses*.

Também concluí-se que para problemas aritméticos com grandes quantidades de dados e que requerem bastante otimização, a linguagem C++ é provavelmente a mais apropriada devido à possibilidade de optimizar o programa de várias formas (embora o Java seja uma boa alternativa). Já o Python seria o menos recomendado para este tipo de problemas, pois como se verificou anteriormente, é muito lento a processar grandes quantidades de dados.

Por último, também se concluiu que a implementação do paralelismo ajudou muito a aumentar a performance do programa, embora tenha sido difícil averiguar qual o número de threads óptimo para obter a melhor performance possível no meu CPU.