U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Performance evaluation of a single core

LEIC - Computação Paralela e Distribuída
3º Ano - 2º Semestre

Turma 5
up201906834 - Óscar Gonçalo Martins Esteves
up201905396 - Pedro Magalhães Moreira Nunes
up201904979 - Luis Carlos Barros Viegas

Performance description and algorithms explanation

No contexto do projeto foi-nos pedido para analisarmos a performance de vários algoritmos de cálculo de matriz. Algoritmos estes que melhoram a performance com base na otimização da cache - queremos reduzir o nível de acesso a níveis de cache maiores.Os 3 algoritmos propostos e desenvolvidos - cálculo matricial normal, cálculo em linha e cálculo com recurso à multiplicação em blocos.

O computador utilizado para medir as performances tem como processador um Intel Core i7-9700 CPU @ 3.00 GHz - 12 MB de cache.

Cálculo matricial normal

O algoritmo do cálculo matricial normal utiliza três ciclos for:
O primeiro serve para percorrer a matriz na horizontal, por todos os elementos dessa linha (segundo a imagem acima, $a_{11} \dots a_{1n}$),

O segundo percorre a matriz na vertical, também por todos os elementos dessa coluna (segundo a imagem acima, $a_{11} \dots a_{n1}$)

Utilizando o terceiro ciclo, o algoritmo vai somando (na variável $temp$) o resultado de $b_{mx} * c_{xn}$, sendo m e n o correspondente a i e j, no algoritmo. Uma vez que os array que contém as matrizes são unidimensionais, como explicado anteriormente, é necessário multiplicar o index i por m_{ar} (tamanho da matriz A e somar k (i é a linha e k é a coluna), bem como para obter os elementos da matriz B é necessário multiplicar k por m_{br} e somar j, (k é a linha e j é a coluna).

A complexidade temporal deste algoritmo é $O(2n^3)$, porém os elementos lidos não são adjacentes, ou seja, a probabilidade de se encontrarem na cache é menor, o que é comprovado pelo hit rate da cache.

```
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;
    }
}
```

```python
for i in range(0, M_AR):
    for j in range(0, M_BR):
        temp = 0
        for k in range(0, M_AR):
            temp += pha[i][k] * phb[k][j]
        phc[i][j] = temp
```

Cálculo matricial em linha

A complexidade temporal é $O(2n^3)$, porém neste método os elementos lidos são adjacentes. Como tal, a probabilidade de eles já estarem na cache para outras operações é muito maior, sendo a hit rate prova disso. Dado isto, apesar de ter a mesma complexidade temporal que o algoritmo normal, possui um tempo total notoriamente mais baixo, pois a maior parte dos elementos necessários está em cache - cache coherence.

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

```python
for j in range(0, M_BR):
    for i in range(0, M_AR):
        temp = 0
        for k in range(0, M_AR):
            temp += pha[i][k] * phb[k][j]
        phc[i][j] = temp
```

Cálculo matricial em bloco

Com este método é utilizado o algoritmo divide and conquer em que dividimos a matriz em pequenas submatrizes e procedemos à multiplicação (em linha) das mesmas.

Com a criação de blocos pequenos, carregamos em cache valores que serão depois reaproveitados, parecido com o método em linha, contribuindo assim para uma elevada hit rate. A diferença é a utilização do block size. Com este block size podemos colocar em cache um valor que definimos em vez de usar um valor igual para tudo ou o espaço total da cache, sendo o ideal um valor que minimize o número de cache misses.

Ou seja, queremos colocar na cache a totalidade do bloco que escolhemos. Por exemplo, se tivermos um block size de 64, temos 32k bytes de informação. Como tal, num cenário idealista, a nossa L1 tem 32k bytes e é ocupada pelo bloco. Com isto evitamos ter que carregar bytes da memória principal, algo bastante mais demorado.

Neste caso temos apenas um core, mas no caso de termos mais poderíamos paralelizar o cálculo das sub-matrizes, aumentando assim a performance.

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

Performance metrics

Foram calculados o tempo de execução e os flops. No caso específico do cpp, e com recurso ao PAPI foram utilizados os eventos das cache misses (L1 e L2) e o numero de stores e loads.

Results and analysis

Multiplicação normal

C++

Size	Time	L1 Data cache misses	L2 Data cache misses	Cache Hit Rate	Gflops
600	0.191	244765168	39509357	0.434856	2.261
1000	1.444	1235799246	285340616	0.383033	1.385
1400	3.864	3520530528	1552384562	0.359194	1.420
1800	18.864	9063645640	8802203510	0.223588	0.618
2200	37.894	17656501116	20595675845	0.171656	0.561
2600	69.464	30874008807	49806833248	0.122376	0.506
3000	113.698	50296911235	95953083365	0.0691975	0.474

Python

Size	Time (seconds)	Gflops
600	37.240	0.0116
1000	204.470	0.00980
1400	613.858	0.00895
1800	1184.823	0.00985
2200	2362.754	0.00901
2600	3795.615	0.00926
3000	5546.097	0.00973

Multiplicação em linha

c++

Size	Time	L1 Data cache misses	L2 Data Cache misses	Cache Hit Rate	Gflops
600	0.102	27102989	58301737	0.958246	4.235

1000	0.490	125747409	261602083	0.958126	4.081
1400	1.623	346291389	708393834	0.957964	3.381
1800	3.516	745917667	1438633098	0.95739	3.317
2200	6.507	2075264285	2590268008	0.935064	3.272
2600	10.658	4413172503	4163108741	0.916335	3.298
3000	16.312	6780618243	6449138544	0.916317	3.310
4096	41.151	17540488675	15944152902	0.914945	3.339
6144	137.783	59118261574	53205674266	0.915052	3.339
8192	328.042	140105271527	127122843525	0.915064	3.351
10240	643.043	279123429381	258172839187	0.915019	3.339

Python

Size	Time (seconds)	Gflops
600	30.464	0.0144
1000	193.699	0.0103
1400	538.915	0.0102
1800	1219.663	0.00956
2200	2109.189	0.0100
2600	3526.110	0.00996
3000	4994.902	0.0108

Multiplicação em bloco

c++

Size	Block Size	Time	L1 Data Cache Misses	L2 Data Cache Misses	Cache Hit Rate
4096	128 256 512	35.287 30.872 41.849	9504448721 9019638257 8782652238	32357706189 23236293452 19070025678	0.954033 0.956318 0.957437
6144	128 256 512	112.940 93.399 92.771	32661300011 30545875028 2964401747	109785966849 79614579977 66280800432	0.953193 0.956164 0.95743
8192	128 256 512	259.157 404.249 349.134	76893909093 72492540937 70566300568	257725698062 164043114541 145852803058	0.953508 0.956109 0.957247
10240	128 256 512	501.580 458.835 448.234	150705053446 141465984849 137092502074	520336858948 360523130651 311806555182	0.953346 0.956146 0.957473

Size	Block Size	Gflops
4096	128 256 512	3.894 4.451 3.284
6144	128 256 512	4.107 4.9663 5.000
8192	128 256 512	4.242 2.719 3.149
10240	128 256 512	4.281 4.680 4.73

Discussão de resultados

Na multiplicação normal, como esperado, o tempo de execução é bastante elevado e a hit rate bastante baixa. A cache não é aproveitada, e com elevados tamanhos de matriz, o overhead da complexidade do algoritmo é notado

Com a multiplicação em linha, a cache é aproveitada, contribuindo assim para elevadas hit-rates (por volta de 90%), tendo tempos notoriamente mais baixos que o algoritmo normal.

No caso da multiplicação em bloco, os valores da hit rate são ligeiramente mais altos (95% para 91%) do que a multiplicação em linha. Tal deve-se ao aproveitamento da memória nos níveis mais rápidos de cache, em vez de simplesmente preenchê-la com a linha toda. Podemos também reparar que não existe linearidade entre a hit rate e os valores do block size. Tal acontece porque idealmente queremos ter um valor de bloco igual ao da cache, para não termos que carregar nada da memória principal.

Um elevado número de FLOPS implica um elevado número de operações em vírgula flutuante, sendo que a multiplicação em bloco possui os maiores valores para o mesmo tamanho de matriz, reforçando uma vez mais o aumento da performance.

Comparação python & cpp

Devido à maneira como python trata da gestão de memória e ao seu cariz interpretado, podemos reparar na disparidade notória entre ambos, enaltecendo assim a relevância do aproveitamento da cache

Conclusão

Comparando os algoritmos utilizados através de várias métricas tais como os Data Cache Misses, o Cache Hit Rate e o tempo que cada algoritmo demora a completar a tarefa, com diferentes block sizes, evidencia-se a importância da Cache.

Apesar de todos os algoritmos terem a mesma complexidade temporal, o facto de os elementos serem computados de forma diferente, e, daí, haver uma maior probabilidade de já se encontrarem numa posição de memória da cache, faz com que o tempo necessário seja muito menor, e que o programa seja muito mais eficiente.