**Q1.4**

A[32][24] - 10002060 h

B[32][24] – 10006060 h

C[32][24] – 1000A060 h

D[32][24] – 1000E060 h

T1[32][24] – 10012060 h

T2[32][24] – 10016060 h

**Q1.6**

Tag – 22 bits

Index – 6 bits

Offset – 4 bits

**Q1.7**

A[32][24] – 10002060 h

Tag – 0001 0000 0000 0000 0010 00

Index – 00 0110

Offset – 0000

B[32][24] – 10006060h

Tag – 0001 0000 0000 0000 0110 00

Index – 00 0110

Offset – 0000

C[32][24] – 1000A060 h

Tag – 0001 0000 0000 0000 1010 00

Index – 00 0110

Offset – 0000

D[32][24] – 1000E060 h

Tag – 0001 0000 0000 0000 1110 00

Index – 00 0110

Offset – 0000

T1[32][24] – 10012060 h

Tag – 0001 0000 0000 0001 0010 00

Index – 00 0110

Offset – 0000

T2[32][24] – 10016060 h

Tag – 0001 0000 0000 0001 0110 00

Index – 00 0110

Offset – 0000

**Q1.8**

O número de vias geralmente diminui o número de misses já que a cache verifica a tag para todas as vias antes de declarar o miss logo quantas mais vias, menor será o número de MISSes a longo prazo.

Por exemplo:

T1[i][j] += A[i][k] \* B[k][j];

Estão se a dar acessos a 3 matrizes, logo se houvesse 2 vias, elas teriam de estar constantemente a ser alteradas, mas como temos 4 vias cada via consegue trabalhar com uma matriz o que diminui o número de MISSes

**Q2.5**

Nº HITS – 256434

Nº MISSES – 284241

Nº ACESSOS – 540675

MISS RATE – 52.57%

**Q2.6**

Tempo médio: 1+0.5257\*100 = 53.57 ns

Tempo total de acesso á memória:

256434+284241\*101 = 28.96 ms

Ou

53.57\*540675 = 28.96 ms

**Q2.7**

Tempo médio s/ cache = 100 ns

Tempo total de acesso á memória s/ cache = 100\*540675 = 54.07 ms

**Q2.8**

A existência da cache faz com que o tempo de acesso á memória seja aproximadamente metade daquele que seria sem cache.

**Q3.2**

Não afeta, pois, vai percorrer ambas as matrizes na mesma, só que com uma ordem diferente

**Q3.3**

A ordem destes ciclos afeta o acesso às caches já que com um dele dá para aproveitar a localidade espacial já que a linha toda é carregada para memória, logo acessos a posições adjacentes são HITs.

**Q3.4**

É a inversa (i,j) pois dá para carregar a linha i para memória e ir de posição j em posição j, só havendo um MISS por linha, muito pelo contrário, se fosse a ordem j,i, haveriam 4 misses por linha já que estamos a aceder a diferentes posições na mesma colunas e sendo assim não conseguimos aproveitar a localidade espacial.

**Q3.5**

Com j,i (ordem original)

Nº Acessos: 540675

Miss rate: 52.57%

Tempo médio: 53.57 ns

Tempo total: 28.96 ms

Com i,j (ordem mais eficiente)

Nº Acessos: 540675

Miss rate: 50.87%

Tempo médio: 51.87 ns

Tempo total: 28.04 ms

**Q3.6**

Sim, esperávamos uma melhoria maior, mas a primeira parte é que apresenta mais misses, por isso não melhorou muito. A primeira parte tem bastantes misses pois tem de aceder a 3 matrizes uma das quais é acedida por colunas, o que não permite uma boa utilização da localidade espacial, logo isso leva a bastantes misses.

**Q4.1**

Função para transposição

# transpõe B e coloca em T2

add t3,zero,s2

add t5,zero,s6

add t0, zero, zero

fploop1:

slli t6,t0,2

add t5,t6,s6

add t1, zero, zero

fploop2:

lw t4,0(t3)

addi t3,t3,4

sw t4,0(t5)

slli t2,s9,2

add t5,t2,t5

addi t1, t1, 1

blt t1, s9, fploop2

addi t0, t0, 1

blt t0, s9, fploop1

# coloca T2 em B

add t3,zero,s2

add t5,zero,s6

add t0, zero, zero

fp1loop1:

add t1, zero, zero

fp1loop2:

lw t4,0(t5)

sw t4,0(t3)

addi t3,t3,4

addi t5,t5,4

addi t1, t1, 1

blt t1, s9, fp1loop2

addi t0, t0, 1

blt t0, s9, fp1loop1

Parte 1

addi t0, zero, 4 #

mul a3, s9, t0 # a3 -> 4\*N

add a0, zero, s1 # a0 -> \*A

add a1, zero, s2 # a1 -> \*B

add a2, zero, s5 # a2 -> \*T1

# for(i=0,...,...){

add t1, zero, zero

f1loop1:

# for(j=0,...,...){

add t2, zero, zero

f1loop2:

# for(k=0,...,...){

add t3, zero, zero

# aux1 = 0

add t4, zero, zero

f1loop3:

# ... += memory(A,R,i,k,0)\*...

lw t5, 0(a0)

# ... += ...\*memory(B,R,k,j,0)

lw t6, 0(a1)

# ... += memory(A,R,i,k,0)\*memory(B,R,k,j,0)

mul t5, t5, t6

# aux1 += memory(A,R,i,k,0)\*memory(B,R,k,j,0)

add t4, t4, t5

# A++ B++

addi a0, a0, 4

addi a1, a1, 4

# for(...,...,k++)

addi t3, t3, 1

# for(...,k<size,...)

blt t3, s9, f1loop3

# }

# memory(T1,W,i,j, aux1)

sw t4, 0(a2)

# T1++

addi a2, a2, 4

# A -= N\*4 (rebobina para o inicio da linha)

sub a0, a0, a3

# for(...,...,j++)

addi t2, t2, 1

# for(...,j<size,...)

blt t2, s9, f1loop2

# }

# A += N\*4 (salta uma linha)

add a0, a0, a3

add a1,zero,s2

# for(...,...,i++)

addi t1, t1, 1

# for(...,i<size,...)

blt t1, s9, f1loop1

# }

A vermelho estão as linhas alteradas

**Q4.3**

Nº HITS – 463874

Nº MISSES – 84993

Nº ACESSOS – 548867

MISS RATE – 15.49%

**Q4.4**

Como as linhas colocadas na cache agora tem valores uteis para o programa o número de misses diminui substancialmente no loop da primeira parte, tomando assim partido da localidade espacial.

**Q4.5**

Tempo médio= 1+0. 1549\*100 = 16.49 ns

Tempo total: 9.05 ms

SpeedUp: 3.09

**Q5.2**

Nº HITS – 428366

Nº MISSES – 120501

Nº ACESSOS – 548867

MISS RATE – 21.95 %

Tempo médio: 22.95 ns

Tempo total: 12.60 ms

SpeedUp: 2.23

**Q5.3**

Como o programa nalgumas ocasiões precisa de rebobinar linhas e de aceder a várias matrizes em simultâneo, o facto de haverem 4 vias permite que haja mais informação com a mesma tag na cache o que facilita o acesso á memória, mas como passa a ser de mapeamento direito isso deixa de acontecer o que leva a uma diminuição da velocidade devido ao aumento dos misses

**Q6.3**

Nº HITS – 334154

Nº MISSES – 222905

Nº ACESSOS – 557059

MISS RATE – 21.90 %

**Q6.4**

Como o nosso código já estava tão optimizado não houve muitas melhorias possíveis a serem feitas, mesmo assim melhorou 0,05%

**Q6.5**

Tempo médio: 22.9 ns

Tempo total: 12.78 ms

**Q6.7**

Tempo médio: 1+0, 2190 \* (20+0,10\*100)= 7.57 ns

Tempo total: 4.21 ms