LAB7

Τζήμας Δημήτρης
Κυριάκος Χρήστος

ΑΣΚΗΣΗ 1

Υποθέσαμε ότι η κύρια μνήμη έχει μέγεθος 512 bytes αρα:

έχουμε διευθυνσιοδότηση 9 bits (2^9 = 512).

Επίσης, επειδή η cache είναι 64 bytes και είναι 2-way set associative με cache block μεγέθους 4 bytes συμπεραίνουμε ότι είναι χωρισμένη σε 8 Blocks (64/4 = 16 → 16/2 = 8) και ότι index set είναι 3 bits (2^3 = 8).

Ακόμα,εφόσον έχουμε 4 bytes/block το offset ειναι 2 (2^2 = 4) και tag είναι 4 bits (9-3-2).

SET INDEX	VALID BIT	TAG	DATA				LRU
			00	01	10	11	
000	1	0000	Mem[0]	Mem[1]	Mem[2]	Mem[3]	0
000	1	1111	Mem[480]	Mem[481]	Mem[482]	Mem[483]	
001	0						0
001	0						
010	0						0
010	0						
011	1	1001	Mem[300]	Mem[301]	Mem[302]	Mem[303]	1
011	1	0011	Mem[108]	Mem[109]	Mem[110]	Mem[111]	
100	0						0
100	0						
101	1	0110	Mem[212]	Mem[213]	Mem[214]	Mem[215]	1
101	0						
110	1	1000	Mem[280]	Mem[281]	Mem[282]	Mem[283]	1
110	0						
111	1	0110	Mem[220]	Mem[221]	Mem[222]	Mem[223]	1
111	0						

ΑΣΚΗΣΗ 2

- Παρατηρώντας την 1η ακολουθία βλέπουμε ότι πρέπει να γίνουν 2 hits στις 6 εισόδους για να έχουμε hit ratio 33%. Τα hit θα γίνουν στις εισόδους 2 και 7 τα οποία θα υπάρχουν στην cache λόγω miss τον 0 και 4, που θα φέρουν από 3 ακόμα bytes με τα επόμενα δεδομένα. Αν αποθηκεύονταν άλλα 7 bytes θα υπήρχε hit και στο 40 λόγω του 32 οπότε θα ανέβαινε το ποσοστό. Άρα 4 συνολικά bytes σε κάθε block άρα 2 bits offset(2^2).

M[0] miss	M[1]	M[2] hit	M[3]
M[4] miss	M[5]	M[6]	M[7] hit

Άρα για 256 bytes cache θα έχουμε 256 /4 = 64 cache blocks και για 512 /4 = 128 cache blocks.

- Στην δεύτερη ακολουθία θέλουμε 1 μόνο hit στις 10 εισόδους. Για direct-mapped θα έχουμε block index log2(128) = 7 bits τα περισσότερα block index bits που μπορούμε να έχουμε(στις άλλες περιπτώσεις θα είναι στάνταρ λιγότερα από 7) και όλα είναι 0 άρα όλα θα μπούν στο ίδιο set. Για 2-way για να γίνει μονο 1 hit θα πρέπει να γίνει στην δεύτερη είσοδο του 1536 επειδή δεν έχει προλάβει να διαγραφεί στην 1η είσοδο του(μεσολαβεί μονο μια είσοδος). Για direct mapped δεν θα έχουμε ποτε hit γιατί πάντα θα αντικαταστατέ και για 4-way θα υπάρχει hit και στο τελευταίο 1024 άρα θα έχουμε 2 hits. Τελικά 2-way associativity.

- Στην τρίτη ακολουθία πρέπει να έχουμε 3 hit στις 10 εισόδους. Εδώ έχει σημασία το set index γιατί τα 7 επόμενα bits μετά τα 2 offset bits διαφέρουν μεταξύ τους άρα θα μπούν σε διαφορετικά sets. Αν έχουμε 256 bits cache τότε 256/4 = 64 cache block και αφού είναι 2-way θα έχουμε 64/2 = 32 sets άρα log2(32) = 5 bits set index και hit θα γίνει στο δεύτερο 256 και στο δεύτερο 64. Ενώ για 512 bytes έχουμε 512/4 = 128 και λόγω 2-way 128/2 = 64 άρα log2(64) = 6 bits set index. Σε αυτήν την περίπτωση έχουμε 3 hit στα δεύτερα 256, 128 και 64.
Άρα , τελικά 512 bytes cache και 6 set index bits.
- Τέλος στην τεταρτη ακολουθία πρέπει να έχουμε 2 hit στις 8 εισόδους. Με FIFO θα έχουμε μονο 1 hit στο τρίτο 0 ενώ με LRU θα έχουμε hit και στο τρίτο και στο τέταρτο 0.

ΑΣΚΗΣΗ 3

- **a:**Λόγω τις τοπικότητας όταν κάνει miss το a[0,0] και το κάθε block έχει μέγεθος 32 bytes αποθηκεύονται το στοιχειο που έκανε miss και αλλα 3 γειτονικά του. Οποτε για 4 loops του **j** έχουμε 3 hits και 1 miss, στα 100 loops έχουμε 25 misses και για 3 loops του **i** είναι : 25*3=75 misses.
- **b:** Όπως και στον πινακα a λόγω τοπικότητας όταν κάνει miss το b[0,0] αποθηκεύονται τα 3 γειτονικά του στοιχεια, δηλαδή το b[0,1], b[0,2], b[1,0].Στις πρώτες 2 διαπεράσεις του b έχουμε 1 miss και 1 hit,ενώ στις επόμενες 4 διαπεράσεις έχουμε 3 hit και 1 miss .Στις 196 διαπεράσεις έχουμε 49 misses + 2= 51 misses σε 1 loop του **i** .Όμως επειδή τα στοιχεια είναι ήδη αποθηκευμένα στην cache στο δεύτερο loop του **i** θεωρούμε ότι όλα κάνουν hit. Συμπεραίνουμε ότι συνολικά για το b πινακα έχουμε 51 misses όλο το χρόνο εκτέλεσης του προγράμματος.
- Συνολικά έχουμε 51+75=126 misses. Σε κάθε loop του j έχουμε 3 διαπεράσεις, σε κάθε loop του **i** έχουμε 300 οποτε συνολικά 300*3=900. Τέλος, miss rate = 126/900 = 14 %.

Loop 1		ΠΡΟΣΠΕΛΑΣΕΙΣ ΤΟΥ **a**	Loop 2	
a[0,0]	M		a[1,0]	M
a[0,1]	H		a[1,1]	H
a[0,2]	H		a[1,2]	H
a[0,3]	H		a[1,3]	H
a[0,4]	M		a[1,4]	M
.			.	
.			.	
.			.	
a[0,99]	H		a[1,99]	H

Loop 1			
b[0,0]	M	b[1,0]	H
b[1,0]	H	b[2,0]	M
b[2,0]	H	b[3,0]	H
b[3,0]	H	b[4,0]	M
b[4,0]	H	b[5,0]	H
.	.	.	.
.	.	.	.
.	.	.	.
b[99,0]	H	b[100,0]	M

ΑΣΚΗΣΗ K-MEANS

A)

InputImage	L1-dcache-loads		L1-dcache-	Branch-misses	cache-references

			load-misses		
andromeda_tiled_rgb.bmp k=4	302713178103		13064789887 4,32% of all L1-dcache hits	2269325657	44706952567
hf_rgb.bmp	322134425303		13606965778 4,22%	4480586728	
Lfh.bmp	226289355888		13525489376	163162131	

B)

Η εκτέλεση του hf_rgb_p2 είναι πιο γρήγορη από την εκτέλεση του hf_rgb_p2p1 επειδή οι στήλες του p2 είναι aligned, δηλαδή είναι δύναμη του 2 (2^15)
και για αυτό αποθηκεύονται στην cache οι τιμές έτσι : πχ για A[3][4]
Cache :

A[0,0]	A[0,1]	A[0,2]	A[0,3]
A[1,0]	A[1,1]	A[1,2]	A[1,3]
A[2,0]	A[2,1]	A[2,3]	A[2,3]

Ενώ ο p2p1 έχει μια παραπάνω στήλη οπότε τα δεδομένα δεν αποθηκεύονται κατάλληλα : πχ A[3][5]
Cache:

A[0,0]	A[0,1]	A[0,2]	A[0,3]
A[0,4]	A[1,0]	A[1,1]	A[1,2]
A[1,3]	A[1,4]	A[2,0]	A[2,1]
A[2,2]	A[2,3]	A[2,4]	A[3,0]
A[3,1]	A[3,2]	A[3,3]	A[3,4]

Αφού το πρόγραμμα διαπερνά τα στοιχεία κατακόρυφα θα θελει να προσπελάσει τα A[0,0]->A[1,0]->A[2,0] κλπ, για αυτό θα εχει παραπάνω cache misses.

C)

1η Βελτίωση: αλλαγή των loop για πρόσβαση πρώτα στην σειρά και μετά στην στήλη για να βοηθήσουμε την cache αφού αποθηκεύει τα γειτονικά στοιχεία στην ίδια σειρά.
2η βελτίωση : αλλαγή του υπολογισμού της απόστασης των σημείων

1. Euclidian distance αλλά βγάζοντας τη ρίζα από το dist
2. Manhattan distance(βλ. Εικ.1):
   Θετικά:Γρηγορότερη από Euclidian distance
   Αρνητικά: Χάνουμε αρκέτα χρώματα αν δεν αυξήσουμε τα clusters.

3. Chebychev distance or Queen-wise distance(βλ. Εικ.2):υλοποιούμε συνάρτηση max(a,b,c) (ideal K = 8)
   Θετικά:Γρήγορη
   Αρνητικά: Χάνουμε αρκέτα χρώματα αν δεν αυξήσουμε τα clusters αλλά λιγότερα από οτι στην Manhattan.

Για την εφαρμογή των παραπάνω κάνουμε το minDist 6300 γιατι εφοσον ειναι γνωστες οι διαστασεις της φωτογραφιας (6000*1918) υπολογιζοντας την διαγωνιο.Παρατηροντας τις 3 αυτες υλοποιησεις συμπεραινουμε οτι μπορουμε σε λιγοτερο χρονο να χρησιμοποιουμε περισσοτερα clusters και να εχουμε κατε επεκταση καλυτερη φωτογραφια μετα το compression Με αποδοτικότερη την metrics του Chebychev(distortion/time).
Συνολικά γλιτώνουμε μια ακριβή πράξη ,τον υπολογισμό της ρίζας.

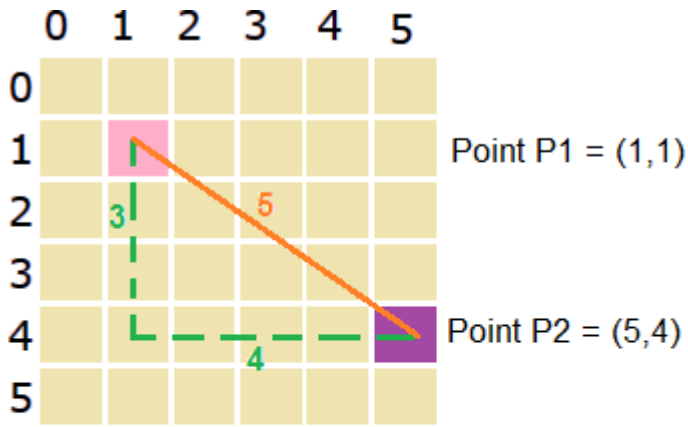
3η βελτίωση : Υλοποίηση δέντρων έτσι ώστε η αρχικοποίηση και η αναζήτηση να γίνουν πιο γρήγορα και να έχουμε όσο το δυνατόν μικρότερο distortion(Loyd's algorithm) (βλ.εικ3).

ΠΗΓΕΣ:
https://www.ingentaconnect.com/content/ist/ei/2016/00002016/00000020/art00036?crawler=true

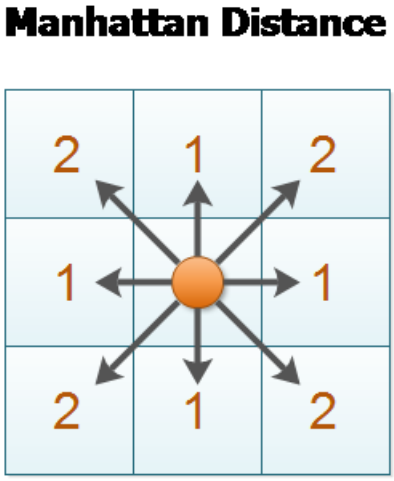
https://www.cs.umd.edu/~mount/Projects/KMeans/pami02.pdf

https://www.researchgate.net/publication/3326408_Fast_tree-structured_nearest_neighbor_encoding_for_vector_quantization

εικ.1



εικ.2



$$|x_1 - x_2| + |y_1 - y_2|$$

εικ.3

```
Filter(kdNode u, CandidateSet Z) {
    C ← u.cell;
    if (u is a leaf) {
        z* ← the closest point in Z to u.point;
        z*.wgtCent ← z*.wgtCent + u.point;
        z*.count ← z*.count + 1;
    }
    else {
        z* ← the closest point in Z to C's midpoint;
        for each (z ∈ Z \ {z*})
            if (z.isFarther(z*, C)) Z ← Z \ {z};
        if (|Z| = 1) {
            z*.wgtCent ← z*.wgtCent + u.wgtCent;
            z*.count ← z*.count + u.count;
        }
        else {
            Filter(u.left, Z);
            Filter(u.right, Z);
        }
    }
}
```