Πανεπιστήμιο Θεσσαλίας
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Lab #8

Ονοματεπώνυμο: Ζαχαριάδης Χαράλαμπος, Ραφτόπουλος Εμμανουήλ

Μάθημα: Οργάνωση και Σχεδίαση Η/Υ – ΑΕΜ: *03734,03735*
Ημερομηνία: *19/1/2024*

Θέμα **1**ο

Απάντηση. Αρχικά θα υπολογίσουμε το μέγεθος των διευθύνσεων του επεξεργαστή, δεδομένου ότι η κύρια μνήμη έχει μέγεθος 512 Bytes. Άρα:

$$2^m = 512 => 2^m = 2^9 => m = 9\ bits$$

Οπότε οι διευθύνσεις έχουν μέγεθος **9 bits**. Έπειτα, θα βρούμε πώς θα οργανωθεί η cache, δηλαδή πόσα *blocks* και *sets* θα έχει.

$$\#block = \frac{cache\_size}{block\_size} = \frac{64}{4} => \#block = 16\ blocks$$

$$\#set = \frac{\#block}{2-way} = \frac{16}{2} => \#set = 2\ sets$$

Ο διαχωρισμός της διεύθυνσης είναι ο εξής:

Bits	m-s-n	s	n
Part Labels	Tag	Set Index	Block Index

$2^n = block\_size => 2^n = 4 => 2^n = 2^2 => n = 2\ bits$
$2^s = \#set => 2^s = 8 => 2^s = 2^3 => s = 3\ bits$

Οπότε η διεύθυνση διαχωρίζεται στα παρακάτω μέρη:

Bits	4	3	2
Part Labels	Tag	Set Index	Block Index

Επείδη τα blocks είναι LRU (Least Recently Used) σε περίπτωση που το set είναι γεμάτο η νέα διεύθυνση παίρνει το block με την πιο παλιά διεύθυνση του set. Η γειτονιά που θα φέρει η επιλεγμένη διεύθυνση μαζί της αποφασίζεται με βάση το block index. Με βάση αυτές τις πληροφορίες έχουμε την παρακάτω προσομοίωση:

Address	Index	Valid	Tag	Data
0110**01**111 Miss	000	N		
		N		
	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [204..207]
		N		
	100	N		
		N		
	101	N		
		N		
	110	N		
		N		
	111	N		
		N		

Address	Index	Valid	Tag	Data
0**110**10100 Miss	000	N		
		N		
	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [204..207]
		N		
	100	N		
		N		
	101	Y	0110	Mem [212..215]
		N		
	110	N		
		N		
	111	N		
		N		

Address	Index	Valid	Tag	Data
1101**00000** Miss	000	Y	1101	Mem [416..419]
		N		
	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [204..207]
		N		
	100	N		
		N		
	101	Y	0110	Mem [212..215]
		N		
	110	N		
		N		
	111	N		
		N		

Address	Index	Valid	Tag	Data
1001**00001**	000	Y	1101	Mem [416..419]
Miss		Y	1001	Mem [288..291]
	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [204..207]
		N		
	100	N		
		N		
	101	Y	0110	Mem [212..215]
		N		
	110	N		
		N		
	111	N		
		N		

Address	Index	Valid	Tag	Data
0110011100	000	Y	1101	Mem [416..419]
Hit		Y	1001	Mem [288..291]
	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [204..207]
		N		
	100	N		
		N		
	101	Y	0110	Mem [212..215]
		N		
	110	N		
		N		
	111	N		
		N		

Address	Index	Valid	Tag	Data
0110111110	000	Y	1101	Mem [416..419]
Miss		Y	1001	Mem [288..291]
	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [204..207]
		N		
	100	N		
		N		
	101	Y	0110	Mem [212..215]
		N		
	110	N		
		N		
	111	Y	0110	Mem [220..223]
		N		

Address	Index	Valid	Tag	Data
1110011101	000	Y	1101	Mem [416..419]
Miss		Y	1001	Mem [288..291]
	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [204..207]
		Y	1110	Mem [460..463]
	100	N		
		N		
	101	Y	0110	Mem [212..215]
		N		
	110	N		
		N		
	111	Y	0110	Mem [220..223]
		N		

Address	Index	Valid	Tag	Data
0011011101	000	Y	1101	Mem [416..419]
Miss		Y	1001	Mem [288..291]
Διώχνω: Mem 204..207	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [108..111]
		Y	1110	Mem [460..463]
	100	N		
		N		
	101	Y	0110	Mem [212..215]
		N		
	110	N		
		N		
	111	Y	0110	Mem [220..223]
		N		

Address	Index	Valid	Tag	Data
1000**11**000	000	Y	1101	Mem [416..419]
Miss		Y	1001	Mem [288..291]
	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [108..111]
		Y	1110	Mem [460..463]
	100	N		
		N		
	101	Y	0110	Mem [212..215]
		N		
	110	Y	1000	Mem [280..283]
		N		
	111	Y	0110	Mem [220..223]
		N		

Address	Index	Valid	Tag	Data
1001**01**111	000	Y	1101	Mem [416..419]
Miss		Y	1001	Mem [288..291]
Διώχνω: Mem 460..463	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [108..111]
		Y	1110	Mem [300..303]
	100	N		
		N		
	101	Y	0110	Mem [212..215]
		N		
	110	Y	1000	Mem [280..283]
		N		
	111	Y	0110	Mem [220..223]
		N		

Address	Index	Valid	Tag	Data
0000**00**010	000	Y	1101	Mem [0..3]
Miss		Y	1001	Mem [288..291]
Διώχνω: Mem 416..419	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [108..111]
		Y	1110	Mem [460..463]
	100	N		
		N		
	101	Y	0110	Mem [212..215]
		N		
	110	Y	1000	Mem [280..283]
		N		
	111	Y	0110	Mem [220..223]
		N		

Address	Index	Valid	Tag	Data
1111**00**000	000	Y	1101	Mem [0..3]
Miss		Y	1001	Mem [480..483]
Διώχνω: Mem 288..291	001	N		
		N		
	010	N		
		N		
	011	Y	0110	Mem [108..111]
		Y	1110	Mem [300..303]
	100	N		
		N		
	101	Y	0110	Mem [212..215]
		N		
	110	Y	1000	Mem [280..283]
		N		
	111	Y	0110	Mem [220..223]
		N		

Θέμα **2**ο

Απάντηση. Αρχικά με τη βοήθεια του κώδικα που μας δίνεται θα αναλύσουμε πώς θα αποθηκεύεται το κάθε στοιχείο του πίνακα στην cache.

```
double a[3][100], b[101][3];

for (i = 0; i < 3; i++)
    for (j =0; j < 100; j++)
        a[i][j] = b[j][0] * b[j+1][0];
```

$sizeof(double) = 8\ Bytes$

$item\_per\_cache\_block = \frac{cache\_block\_size}{sizeof(double)} = \frac{32}{8} = 4\ items$

Πίνακας a:

a[0][0]	a[0][1]	a[0][2]	...	a[0][98]	a[0][99]
a[1][0]	a[1][1]	a[1][2]	...	a[1][98]	a[1][99]
a[2][0]	a[2][1]	a[2][2]	...	a[2][98]	a[2][99]

Πίνακας b:

b[0][0]	b[0][1]	b[0][2]
b[1][0]	b[1][1]	b[1][2]
b[2][0]	b[2][1]	b[2][2]
...	...	...
b[j][0]	b[j][1]	b[j][2]
b[j+1][0]	b[j+1][1]	b[j+1][[2]]
...	...	...
b[100][0]	b[100][1]	b[100][2]

Επειδή ο πίνακας a θα αποθηκεύεται στην cache ανά τετράδες αρχίζοντας από το 0, μόνο για διαπεράσεις του j πολλαπλάσιες του 4 π.χ. 0,4,8,..,96. Άρα στις 100 διαπεράσεις **miss** θα έχουν οι $100/4 = 25$ διαπεράσεις του πίνακα α. Όμως το πρόγραμμά μας κάνει $3 * 100 = 300$ επαναλήψεις. Αυτό σημαίνει ότι εξαιτίας του πίνακα α θα έχουμε $3 * 25 =$ **75 misses**.

Όσον αφορά τον πίνακα b μόνο στην πρώτη εξωτερική επανάληψη θα υπάρχουν **misses** καθώς από εκεί και έπειτα θα προσπελαύνονται τα ίδια στοιχεία. Συγκεκριμένα, στην πρώτη εκτέλεση της εσωτερικής επανάληψης για j = 0 θα υπάρξουν 2 **misses** για b[0][0] και για b[1][0], από εκεί και έπειτα, όμως, το στοιχείο b[j][0] θα έχει διαπεραστεί από την προηγούμενη επανάληψη ως το b[j+1][0] στοιχείο και άρα θα κάνει **miss** μόνο το b[j+1][0] στοιχείο αυτής της επανάληψης. Κατά συνέπεια η προσπέλαση του πίνακα b δημιουργεί $2 + 99 =$ **101 misses** από συνολικά $2(indexes\ read\ per\ loop) * 3(i\ loop) * 100(j\ loop) = 600$ φορές που θα γίνει **read**.

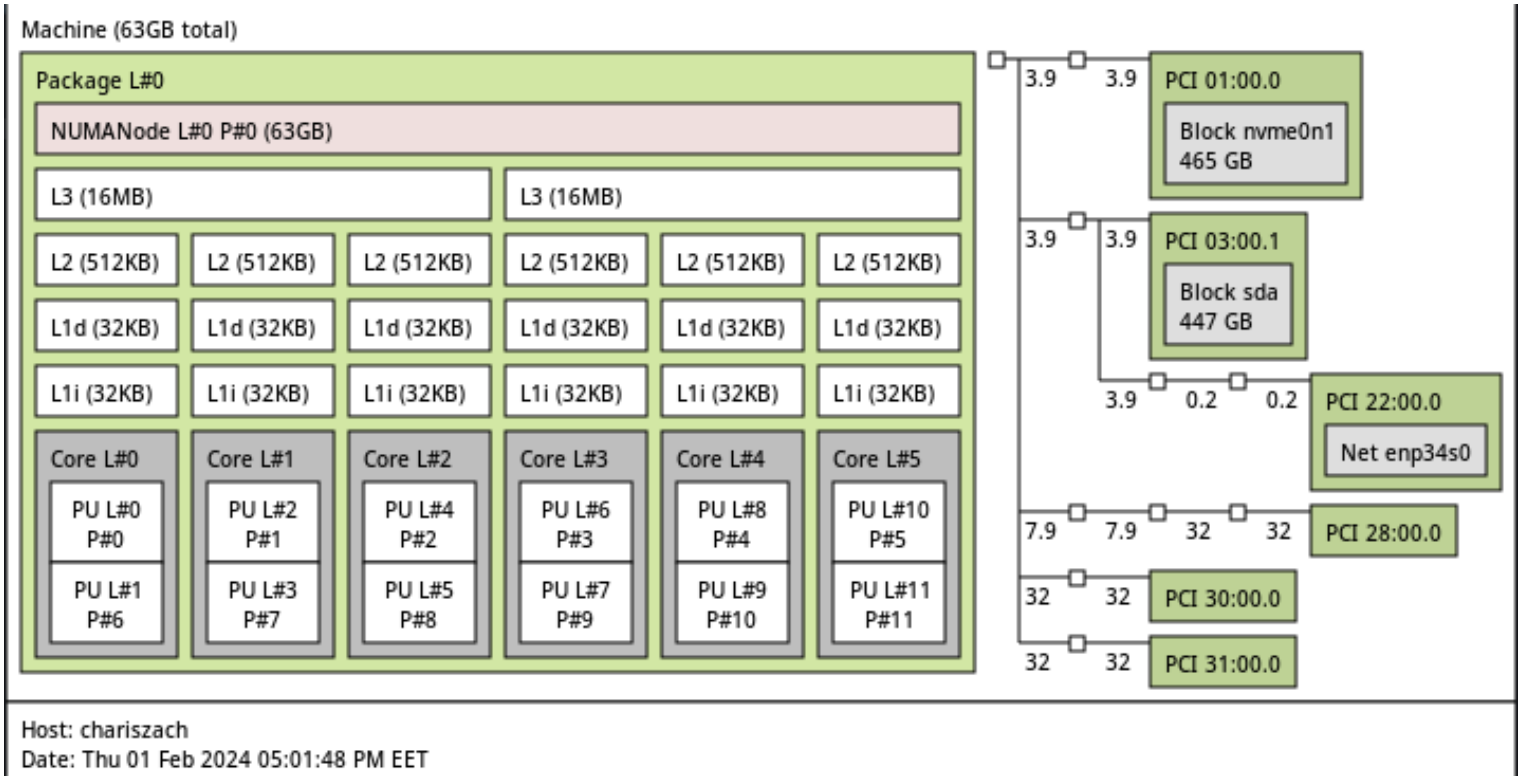
Τελικά, η εκτέλεση του προγράμματος δημιουργεί συνολικά 75 + 101 = **176 misses** και έχει miss rate $\frac{176}{300+600} =$ **19.55%**.

Θέμα **3**ο

Τα specifications του υπολογιστή στον οποίο θα τρέχουμε τα πειράματα:

RAM	63GB
Cores	6
Clock Speed	4 GHz
L3	32MB
L2	3MB
L1	384KiB

Τοπολογία CPU, με όνομα μοντέλου **"AMD Ryzen 5 3600 6-Core Processor"**:



Απάντηση**.** α) Αρχίζοντας το πείραμα με K **= 4**:

	Andromeda Tiled	HF	LFH	LFV
Duration	182s	199s	173s	128s
Cycles	710.9M	778M	672.8M	659.1M
Instructions	699.2M	698.8M	699.3M	698.6M
Instructions Per Cycle	0.98	0.90	1.04	1.06
Stalled Cycles Frontend	96M (13.48% idle)	79.6M (10.24% idle)	33.8M (5.04% idle)	148.4M (22.52% idle)
Stalled Cycles Backend	353.6M (49.74% idle)	275.1M (35.36%)	459.6M (68.32% idle)	335.9M (50.97% idle)
L1-Dcache Loads	377.3B	433.2M	304.1M	299.7M
L1-Dcache Misses	13.5M (3.59%)	13.5M (3.12%)	13.5M (4,44%)	13.5M (4.52%)
L1-Dcache Prefetches	54k	48.7k	55.4k	54.8k
L1-Icache loads	4.9M	5.2M	6.1M	4.9M
L1-Icache misses	58k (1.16%)	64.2k (1.23%)	57.2k (0.94%)	59.6k (1.2%)
Cache References	45M	42.8M	43.7M	45.6M
Cache Misses	24.3M (53.94%)	25.2M (58.81%)	24.8M (56.80%)	24M (52.72%)
Branch loads	128.7M	128.7M	128.7M	128.7M
Branch misses	1.8M (1.46%)	4.1M (3.23%)	160.7k (0.12%)	108.8M (0.08%)

Ξανακάνουμε το πείραμα με K **= 2**:

	Andromeda Tiled	HF	LFH	LFV
Duration	121s	149s	120s	91s
Cycles	467.4M	579.6M	464.3M	354.5M
Instructions	544.7M	567.6M	388.8M	435.3M
Instructions Per Cycle	1.17	0.98	0.82	1.23
Stalled Cycles Frontend	38.4M (8.23% idle)	21.2M (3.67% idle)	14.3M (3.10% idle)	37.8M (10.67% idle)
Stalled Cycles Backend	273.2M (58.46% idle)	273.8M (47.24% idle)	342.6M (73.79% idle)	218M (61.51% idle)
L1-Dcache Loads	302.2M	399.5M	221.2M	212.3M
L1-Dcache Misses	9.5M (3.16%)	9.8M (2.47%)	7.3M (3.32%)	7.3M (3.36%)
L1-Dcache Prefetches	47.2k	61.5k	30.7k	28.9k
L1-Icache loads	4.2M	4.2M	3.4M	3M
L1-Icache misses	47.4k (1.12%)	54.1k (1.27%)	39.8k (1.14%)	35.3k (1.14%)
Cache References	34.9M	35.4M	25.3M	26.4M
Cache Misses	16.5M (47.32%)	17.9M (50.54%)	13.3M (52.61%)	12.5M (47.55%)
Branch loads	102.2M	106.6M	71.2M	81.5M
Branch misses	547.3k (0.54%)	2.7M (2.54%)	152.7k (0.21%)	30.8k (0.05%)

Τέλος, εκτελούμε το πείραμα με K **= 8**:

	Andromeda Tiled	HF	LFH	LFV
Duration	279s	307s	243s	265s
Cycles	1.1M	1.2B	948.4M	1B
Instructions	958.8M	958.3M	959.3M	959.6M
Instructions Per Cycle	0.88	0.80	1.01	0.92
Stalled Cycles Frontend	248.6M (22.74% idle)	283.3M (23.56% idle)	174M (18.35% idle)	328.4M (31.65% idle)
Stalled Cycles Backend	469.6M (42.96% idle)	400.9M (33.34% idle)	540.4M (56.98% idle)	446.6M (43.05% idle)
L1-Dcache Loads	437.4M	448.4M	343.9M	341.8M
L1-Dcache Misses	21M (4.81%)	20.9M (4.67%)	20.9M (6.08%)	21M (6.16%)
L1-Dcache Prefetches	86.8k	73.1k	64.2k	113.2k
L1-Icache loads	6.5M	8.5M	9.6M	6.4M
L1-Icache misses	76.5k (1.17%)	88.5k (1.84%)	84.7k (0.88%)	81.4k (1.26%)
Cache References	64.9M	62.9M	63.9M	66.5M
Cache Misses	37.1M (57.21%)	38.1M (60.64%)	36.9M (56.41%)	37.5M (56.41%)
Branch loads	172.9M	172.9M	172.9M	172.8M
Branch misses	2.6M (1.50%)	4.9M (2.89%)	137.8k (0.08%)	225.7k (0.13%)

β) Ομοίως, αρχίζουμε το νέο πείραμα με K **= 4**:

	hf_rgb_p2	hf_rgb_p2p1
Duration	268s	144s
Cycles	1B	558.4M
Instructions	509.5M	509.2M
Instructions Per Cycle	0.48	0.91
Stalled Cycles Frontend	117.7M (11.20% idle)	52.9M (9.48% idle)
Stalled Cycles Backend	577.3M (54.90% idle)	194.2M (34.79% idle)
L1-Dcache Loads	337.7M	316.7M
L1-Dcache Misses	10.3M (3.06%)	9.9M (3.15%)
L1-Dcache Prefetches	58.3k	110.5k
L1-Icache loads	4.6M	3.5M
L1-Icache misses	43.9k (0.95%)	42.3k
Cache References	27.6M	27.7M
Cache Misses	17.9M (64.74%)	17.4M (62.80%)
Branch loads	93.8M	93.8M
Branch misses	3.1M (3.31%)	3M (3.21%)

Συνεχίζουμε το πείραμα με K **= 2**:

	hf_rgb_p2	hf_rgb_p2p1
Duration	229s	108s
Cycles	897.2M	419.6M
Instructions	413.4M	413.7M
Instructions Per Cycle	0.46	0.99
Stalled Cycles Frontend	33.6M (3.75%)	11.8M (2.82% idle)
Stalled Cycles Backend	623.4M (69.48%)	203.8M (48.58% idle)
L1-Dcache Loads	303.8M	299.3M
L1-Dcache Misses	7.4M (2.45%)	7.2M (2.43%)
L1-Dcache Prefetches	84k	132.7k
L1-Icache loads	3.5M	2.9M
L1-Icache misses	33.7k (0.95%)	34.8k (1.17%)
Cache References	21.2M	21.7M
Cache Misses	12.6M (59.44%)	14.1M (64.92%)
Branch loads	77.6M	77.7M
Branch misses	1.9M (2.52%)	2M (2.60%)

Ολοκληρώνουμε το πείραμα με **K = 8**:

	hf_rgb_p2	hf_rgb_p2p1
Duration	344s	220s
Cycles	1.3B	857.4M
Instructions	698.6M	698.5M
Instructions Per Cycle	0.52	0.81
Stalled Cycles Frontend	324.8M (24.14% idle)	205.8M (24.01%)
Stalled Cycles Backend	616.7M (45.83% idle)	289.5M (33.75%)
L1-Dcache Loads	325.6M	320.1M
L1-Dcache Misses	16.6M (5.10%)	15.5M (4.85%)
L1-Dcache Prefetches	71.2k	162.7k
L1-Icache loads	7.7M	5.9M
L1-Icache misses	66.3k (0.86%)	59.8 (1.01%)
Cache References	41.5M	41.7M
Cache Misses	28.7M (69.25%)	25M (59.99%)
Branch loads	126M	126M
Branch misses	3.4M (2.73%)	3.4M (2.76%)

Σχολιασμός. Αρχικά, ας εξηγήσουμε τη σημασία μερικών παραμέτρων των πινάκων:

- **Stalled Cycles Frontend:** Αναφέρεται στα stalls που έγιναν στο frontend της CPU, όπου εκεί γίνεται instructuction fetch και instruction decode.
- **Stalled Cycles Backend:** Αναφέρεται στα stalls που έγιναν στο backend της CPU, όπου εκεί γίνονται τα υπόλοιπα operations της CPU.
- **Cache References:** Μετράει της φορές που ζητήθηκαν δεδομένα από την cache.

1. Πρώτα από όλα, από όλες αυτές τις μετρήσεις παρατηρούμε ότι ο κώδικας είναι υπερβολικά αργός. Για παράδειγμα, ας εστιάσουμε στο πρώτο πείραμα με την εικόνα "andromeda_tiled_rgb". Εκεί παρατηρούμε ότι ο χρόνος εκτέλεσης είναι 182s (περίπου 1,5 λεπτό!). Εμβαθύνοντας την ανάλυσή μας παρατηρούμε ότι ο εξαπύρηνος CPU πραγματοποιεί μόλις 0.98 εντολές ανά κύκλο, να σημειωθεί ότι για τους σύγχρονους επεξεργαστές η μέση τιμή των Instructions Per Cycle (IPC) είναι περίπου 4 . Από τις 700M εντολές οι 129M είναι branch loads όπου από αυτά το 1.46 % είναι branch misses. Αυτό οδηγεί σε μια πληθώρα από stalls στο frontend και στο backend της CPU (96M frontend stalls, 353M backend stalls). Ακομά, παρατηρούμε ότι το υπάρχει πολύ μεγάλο ποσοστό cache misses (53.94 %) και επειδή από τα δεδομένα της L1 cache παρατηρούμε ότι τα misses της D-Cache είναι πολύ μεγαλύτερα από της I-Cache (13.2 M έναντι των 58k) μεγαλύτερη ζημιά γίνεται από την ανάρτηση δεδομένων από την cache. Τέλος τα backend stalls είναι σχεδόν τετραπλάσια από τα frontend, που δηλώνει ότι υπάρχουν επιπλέον πολλές εγγραφές σε καταχωρητή που οδηγούν σε επιπλέον stalls ώστε το backend να εκτελέσει σωστά τα απαραίτητα operations.

   Παραμένωντας στην ίδια εικόνα αλλά υποδιπλασιάζοντας ή διπλασιάζοντας τον αριθμό των clusters βγάζουμε μερικά νέα συμπεράσματα. Αρχικά, βλέπουμε ότι υπάρχει μια σημαντική μείωση του χρόνου εκτέλεσης όταν K = 2, αλλά και μια ακόμα πιο σημαντική αύξηση όταν K = 8, που είναι λογικό αφού όσο περισσότερα clusters βάλουμε τόσο περισσότερες εντολές θα εκτελεστούν. Επιπλεόν, τα IPC τείνουν να αυξάνονται όσο μικραίνονται τα clusters εκτός από την "lfh_rgb" όπου εκεί μέγιστο IPC εμφανίζει για K = 4. Τα cache misses όσο μειώνεται το K τόσο πιο πολύ μειώνεται και το ποσοστό τους. Τέλος, τα branch misses τείνουν να έχουν μικρότερο ποσοστό για K = 2, χωρίς όμως αυτό να συμβαίνει αυστηρά για κάθε εικόνα.

   Παρατηρούμε πως ο χρόνος εκτέλεσης είναι μεγαλύτερος στην "hf" όπου εκεί παρατηρούμε ότι ο χρωματισμός των pixels είναι εντελώς τυχαία κατανεμημένος (high entropy) και ποίκιλος) δυσκολεύοντας τη λειτουργία του kmeans και πιο μικρούς χρόνους εκτέλεσης έχουμε στις "lfh" και "lfv" όπου εκεί δεν υπάρχει μεγάλη ποικιλία χρωμάτων και τα pixels είναι αρκετά κατανεμημένα (low entropy).

   Μια άλλη παρατήρηση που πρέπει να γίνει είναι στις εικόνες "lfh" και "lfv". Εκεί ο κώδικας εκτελείται πολύ πιο γρήγορα στην "lfv" που οι λωρίδες είναι κατά στήλη, σε σχέση με τη "lfh" που οι λωρίδες είναι κατά γραμμή. Αυτό συμβαίνει επειδή ο κώδικας τρέχει ανά στήλη και όχι ανα γραμμή και ομαδοποιεί πιο εύκολα τις λωρίδες που είναι σε σχήμα στηλών με τη βοήθεια της cache.

2. Σχετικά με το δεύτερο πείραμα έχουμε δύο εικόνες την hf_rgb_p2 και hf_rgb_p2p1. Η μόνη τους διαφορά είναι μία επιπλέον στήλη με pixel. Παρ' όλα αυτά η διαφορά τόσο στο χρόνο εκτέλεσης όσο και στους κύκλους stall είναι μεγάλη, κάτι που ίσως φαίνεται περίεργο εφόσον η 2η εικόνα είναι μεγαλύτερη από την 1η. Το γεγονός αυτό συμβαίνει, δίοτι στην 1η εικόνα όπου το width είναι 2^{15} (ευθυγράμμιση της cache) έχουμε αρκετές αστοχίες δίενεξης (conflicts) και πολλά μπλοκ προσπαθούνε να μπουν στο ίδιο set της cache. Από την άλλη η 2η εικόνα δεν είναι ευθυγραμμισμένη και για το λόγο αυτό έχουμε διπλάσια prefetches που τελικά μειώνουν το χρόνο εκτέλεσης του προγράμματος σχεδόν κατά το ήμισυ.

γ) **Optimization** Κώδικα **KMeans**

Αφού τρέξαμε τον αρχικό κώδικα με όλες τις παραπάνω εικόνες με τη χρήση του perf (linux-tools) και του vtune-gui (intel-oneapi) παρατηρήσαμε μερικά σημεία "κλειδιά" στα οποία θα μπορούσαμε να κάνουμε αποτελεσματικές αλλαγές. Συγκεκριμένα:

1. Βάλαμε σε εντολές διακλάδωσης που γνωρίζαμε την συμπεριφορά τους, τον preprocessor να περιμένει να γίνονται πάντα taken ή not taken με τη χρήση του __builtin_expect().

2. Αλλάξαμε τα for loops, κυρίως τα εμφωλευμένα όπου στην εξωτερική επανάληψη είχαν τις στήλες (cols) της εικόνας και στην εσωτερική τις γραμμές (rows) και τις αναστρέψαμε ώστε να αυξηθεί η χωρική τοπικότητα του προγράμματος.

3. Τους counters των for loop τους ορίσαμε ως registers για να γλιτώσουμε σημαντικό χρόνο ($\approxeq$ 7 δευτερόλεπτα)

4. Αφαιρέσαμε μερικές γραμμές κώδικα (αχρείαστες μεταβλητές που έκαναν αχρείαστες αλλά "βαριές" για τον επεξεργαστή πράξεις, όπως η μεταβλητή pixel).

5. Τον υπολογισμό της απόστασης σε ένα σύστημα (x,y,z) όπως το δικό μας τον απλοποίησαμε, αφού δεν μας ενδιαφέρει η ακρίβεια της πράξης από: $\sqrt{((r - means[i].r)^2 + (g - means[i].g)^2) + (b - means[i].b)^2}$ σε $|r + g + b - (means[i].r + means[i].g + means[i].b)|$ επειδή κάνουμε μόνο συγκρίσεις με αυτές τις τιμές.

6. Μεταβλητές όπως noOfMoves που δεν υπήρχε λόγος να είναι unsigned long int (παίρνει τιμές 0 και 1) τις κάναμε int.

7. Προσπαθήσαμε μικρές if που ήταν για ανάθεση τιμών σε συγκεκριμένες μεταβλητές να τις μετατρέψουμε σε ternary operators (? :). Εφαρμόζωντας το στη if της minDist, που είναι ένα από τα hotspots του κώδικα, γλυτώσαμε μέχρι και 10 δευτερόλεπτα εκτέλεσης.

8. Βάλαμε το -Ofast flag στο compile.

9. Μια άλλη καλή κίνηση είναι να μειώσουμε τον αριθμό των max iterations, ώστε να χρειαστούν λιγότερες επαναλήψεις για να τερματίσει ο κώδικας (π.χ. από 10 σε 8). Αυτό έχει σημαντικό όφελος στον χρόνο εκτέλεσης και η επίπτωση στην ποιότητα της εικόνας εξόδου είναι αρκετά μικρή.

Εκτός αυτών θα μπορούσαμε να αξιοποιήσουμε παραπάνω threads του επεξεργαστή δουλεύοντας παράλληλα ώστε να μειωθεί ραγδαία ο χρόνος εκτέλεσης. Προσπαθήσαμε να κάνουμε χρήση της βιβλιοθήκης openmp και του pragma, αλλά χωρίς κάποιο αξιόλογο αποτέλεσμα.