Άσκηση 1

Θεωρούμε το ακόλουθο κομμάτι κώδικα

```
int i,j;
double result, a[110][4];

for(i=0; i<4; i++)
for(j=0; j<100; j++)
    result += a[j][i]*a[j+1][i] + 0.5;
```

Υποθέσεις:
- κάθε στοιχείο του πίνακα έχει μέγεθος 8 bytes

- υπάρχει 1 επίπεδο κρυφής μνήμης, πλήρως συσχετιστικής, με LRU πολιτική αντικατάστασης, αποτελούμενη από 100 blocks δεδομένων
- το μέγεθος του block είναι 32 bytes

- ο πίνακας είναι αποθηκευμένος στην κύρια μνήμη κατά γραμμές, και είναι «ευθυγραμμισμένος» ώστε το 1ο στοιχείο του να απεικονίζεται στην αρχή μιας γραμμής της cache

- αρχικά η cache είναι άδεια

- *Βρείτε ποιες από τις αναφορές στα στοιχεία του πίνακα a για όλη την εκτέλεση του παραπάνω κώδικα καταλήγουν σε misses στην cache.*

- *Υποδείξτε ποια είναι compulsory, ποια είναι capacity, και ποια conflict.*

- *Δώστε τον συνολικό αριθμό των misses.*

- 1 block = 32 bytes
- 1 στοιχείο = 8 bytes
- πίνακας αποθηκευμένος κατά γραμμές

→ σε 1 block της cache θα απεικονίζονται 4 διαδοχικά στοιχεία του πίνακα, π.χ.

a[i][j], a[i][j+1], a[i][j+2], a[i][j+3]

επιπλέον

- πίνακας ευθυγραμμισμένος

→ <u>σε 1 block της cache θα απεικονίζεται 1 ολόκληρη γραμμή του πίνακα</u>, δηλαδή

a[i][j], a[i][j+1], a[i][j+2], a[i][j+3]

όπου j%4=0

αναφορά στη μνήμη

περιεχόμενα cache

i=0,j=0

a[0][0]

compulsory miss

δηλαδή, αναφερόμαστε για 1η φορά στα αντίστοιχα blocks…

a[0][0]	a[0][1]	a[0][2]	a[0][3]

αναφορά στη μνήμη

περιεχόμενα cache

i=0,j=0

a[0][0] compulsory miss

a[1][0] compulsory miss

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]

αναφορά στη μνήμη

περιεχόμενα cache

i=0,j=0

a[0][0] compulsory miss

a[1][0] compulsory miss

i=0,j=1

a[1][0] hit

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]

αναφορά στη μνήμη

περιεχόμενα cache

i=0,j=0

a[0][0] compulsory miss

a[1][0] compulsory miss

i=0,j=1

a[1][0] hit

a[2][0] compulsory miss

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][2]	a[2][3]

αναφορά στη μνήμη

περιεχόμενα cache

i=0,j=0

a[0][0] compulsory miss

a[1][0] compulsory miss

i=0,j=1

a[1][0] hit

a[2][0] compulsory miss

Έτσι για i=0, j=2…99 θα έχουμε 1 hit + 1 comp. miss για κάθε επανάληψη του j

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][2]	a[2][3]

αναφορά στη μνήμη

περιεχόμενα cache

i=0,j=0

a[0][0] compulsory miss

a[1][0] compulsory miss

i=0,j=1

a[1][0] hit

a[2][0] compulsory miss

Έτσι για i=0, j=2…99 θα έχουμε 1 hit + 1 comp. miss για κάθε επανάληψη του j

για i=0 θα έχουμε 2+99=101 misses, όλα compulsory

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][2]	a[2][3]

αναφορά στη μνήμη

περιεχόμενα cache

i=0,j=99

a[99][0] compulsory miss

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][2]	a[2][3]
a[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
…	…	…	…
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

αναφορά στη μνήμη

περιεχόμενα cache

i=0,j=99

a[99][0] compulsory miss

a[100][0] compulsory miss

αντικατέστησε το LRU block που υπήρχε στην cache

a[100][0]	a[100][1]	a[100][2]	a[100][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
a[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
…	…	…	…
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

αναφορά στη μνήμη

περιεχόμενα cache

i=1,j=0

a[0][1]

capacity miss

διότι αντικαταστάθηκε λόγω έλλειψης χώρου το block που είχε έρθει στην cache κατά το παρελθόν και το περιείχε

a[100][0]	a[100][1]	a[100][2]	a[100][3]
a[0][0]	**a[0][1]**	a[0][2]	a[0][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]
a[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
…	…	…	…
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

αναφορά στη μνήμη

περιεχόμενα cache

i=1,j=0

a[0][1] capacity miss

a[1][1] capacity miss

a[100][0]	a[100][1]	a[100][2]	a[100][3]
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[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
…	…	…	…
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

αναφορά στη μνήμη

i=1,j=0

a[0][1] capacity miss

a[1][1] capacity miss

i=1,j=1

a[1][1] hit

περιεχόμενα cache

a[100][0]	a[100][1]	a[100][2]	a[100][3]
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[3][0]	a[3][1]	a[3][2]	a[3][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
…	…	…	…
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

αναφορά στη μνήμη

περιεχόμενα cache

i=1,j=0

a[0][1] capacity miss

a[1][1] capacity miss

i=1,j=1

a[1][1] hit

a[2][1] capacity miss

a[100][0]	a[100][1]	a[100][2]	a[100][3]
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][2]	a[2][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
…	…	…	…
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

αναφορά στη μνήμη

περιεχόμενα cache

i=1,j=0

a[0][1] capacity miss

a[1][1] capacity miss

i=1,j=1

a[1][1] hit

a[2][1] capacity miss

a[100][0]	a[100][1]	a[100][2]	a[100][3]
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][2]	a[2][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
…	…	…	…
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

Στις επόμενες επαναλήψεις θα έχουμε κυκλικές αντικαταστάσεις blocks, οπότε τα misses και τα hits θα ακολουθούν το ίδιο μοτίβο όπως και για i=0.

αναφορά στη μνήμη

περιεχόμενα cache

i=1,j=0

a[0][1] capacity miss

a[1][1] capacity miss

i=1,j=1

a[1][1] hit

a[2][1] capacity miss

a[100][0]	a[100][1]	a[100][2]	a[100][3]
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][2]	a[2][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
…	…	…	…
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

Στις επόμενες επαναλήψεις θα έχουμε κυκλικές αντικαταστάσεις blocks, οπότε τα misses και τα hits θα ακολουθούν το ίδιο μοτίβο όπως και για i=0.

↓

συνολικά θα έχουμε 4*101=404 misses

αναφορά στη μνήμη

περιεχόμενα cache

i=1,j=0

a[0][1] capacity miss

a[1][1] capacity miss

i=1,j=1

a[1][1] hit

a[2][1] capacity miss

a[100][0]	a[100][1]	a[100][2]	a[100][3]
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][2]	a[2][3]
a[4][0]	a[4][1]	a[4][2]	a[4][3]
…	…	…	…
a[98][0]	a[98][1]	a[98][2]	a[98][3]
a[99][0]	a[99][1]	a[99][2]	a[99][3]

Conflict misses δεν έχουμε διότι η cache είναι fully associative → τα blocks δεδομένων μπορούν να απεικονιστούν οπουδήποτε στην cache

*Το σύνολο εντολών της αρχιτεκτονικής του επεξεργαστή διαθέτει μία ειδική εντολή **prf(*addr)**. Η εντολή αυτή προ-φορτώνει στην κρυφή μνήμη ολόκληρο το μπλοκ που περιέχει τη λέξη που βρίσκεται στη διεύθυνση μνήμης addr.*

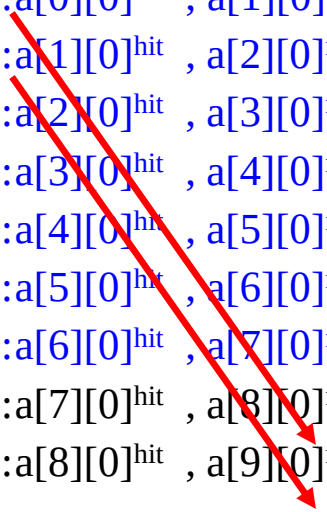
- Χωρίς να αλλάξετε τη σειρά των loads για τις αναφορές στα στοιχεία του πίνακα, εισάγετε κλήσεις στην prf (1 ή περισσότερες) στον παραπάνω κώδικα ώστε να μειωθούν τα misses. Δώστε τον συνολικό αριθμό των misses.

Υποθέστε ότι 7 επαναλήψεις του εσωτερικού loop είναι αρκετές ώστε να "καλυφθεί" ο χρόνος που απαιτείται για να έρθουν τα δεδομένα που ζητά η prf στην cache.

Μη λάβετε ειδική μέριμνα για την προφόρτωση δεδομένων στις αρχικές επαναλήψεις του loop, ούτε για τις έξτρα προφορτώσεις στις τελευταίες επαναλήψεις.

Ας δούμε τις αναφορές σε στοιχεία του πίνακα για τις πρώτες επαναλήψεις του loop:

(i,j)=
(0,0):a[0][0]^{miss}, a[1][0]^{miss}
(0,1):a[1][0]^{hit} , a[2][0]^{miss}
(0,2):a[2][0]^{hit} , a[3][0]^{miss}
(0,3):a[3][0]^{hit} , a[4][0]^{miss}
(0,4):a[4][0]^{hit} , a[5][0]^{miss}
(0,5):a[5][0]^{hit} , a[6][0]^{miss}
(0,6):a[6][0]^{hit} , a[7][0]^{miss}
(0,7):a[7][0]^{hit} , a[8][0]^{miss}
(0,8):a[8][0]^{hit} , a[9][0]^{miss}



- οι πρώτες επαναλήψεις που χρειάζονται για να καλύψουν χρονικά τη μεταφορά δεδομένων που ζητά η prf
- δηλαδή, αν η prf κληθεί στην (0,0), τότε τα δεδομένα που ζήτησε θα έρθουν στην (0,7)

για ποιες αναφορές θα εφαρμόσουμε προφόρτωση;

για εκείνες που καταλήγουν σε misses (a[j+1][i])

έτσι, στην επανάληψη (0,0) πρέπει να ζητήσουμε το a[8][0], στην (0,1) το a[9][0], κ.ο.κ.

```
for(i=0; i<4; i++)
for(j=0; j<100; j++) {
    prf(&a[j+8][i]);
    result += a[j][i]*a[j+1][i] + 0.5;
}
```

Έτσι, τα μόνα misses που συμβαίνουν σε κάθε επανάληψη είναι τα εξής:

(i,j)=

(0,0):a[0][0]^{miss}, a[1][0]^{miss}

(0,1):a[1][0]^{hit} , a[2][0]^{miss}

(0,2):a[2][0]^{hit} , a[3][0]^{miss}

(0,3):a[3][0]^{hit} , a[4][0]^{miss}

(0,4):a[4][0]^{hit} , a[5][0]^{miss}

(0,5):a[5][0]^{hit} , a[6][0]^{miss}

(0,6):a[6][0]^{hit} , a[7][0]^{miss}

(0,7):a[7][0]^{hit} , a[8][0]^{hit}

(0,8):a[8][0]^{hit} , a[9][0]^{hit}

} συνολικά, θα έχουμε 4*8=32 misses

Υποθέστε τώρα ότι έχετε μια cache ίδιας οργάνωσης, αλλά "απείρου" μεγέθους.

- Πώς θα ξαναγράφατε τον κώδικα του ερωτήματος 2, μειώνοντας περαιτέρω τον αριθμό των κλήσεων στην prf;

για να ελαχιστοποιήσουμε τα prefetches, εκτελούμε την 1η μόνο επανάληψη του εξωτερικού βρόχου με prefetches και τις υπόλοιπες ως έχουν

ξέρουμε ότι η cache είναι "απείρου" μεγέθους, οπότε οι επόμενες επαναλήψεις δεν οδηγούν σε capacity misses

```
for(j=0; j<100; j++){
prf(&a[j+8][0]);
result += a[j][0]*a[j+1][0] + 0.5;
}

for(i=1; i<4; i++)
for(j=0; j<100; j++)
    result += a[j][i]*a[j+1][i] + 0.5;
```

συνολικά, θα έχουμε 8 misses

Θεωρείστε πάλι τον αρχικό κώδικα (χωρίς τις προφορτώσεις). Εκτός από την προφόρτωση δεδομένων, ποια άλλη γνωστή τεχνική βελτιστοποίησης θα εφαρμόζατε στον κώδικα ώστε να μειωθούν τα misses;

```
for(i=0; i<4; i++)
for(j=0; j<100; j++)
    result += a[j][i]*a[j+1][i] + 0.5;
```

στο σώμα του loop δεν υπάρχουν εξαρτήσεις, επομένως μπορούμε να εφαρμόσουμε *αναδιάταξη βρόχων (loop interchange)*

```
for(j=0; j<100; j++)
for(i=0; i<4; i++)
    result += a[j][i]*a[j+1][i] + 0.5;
```

τώρα ο πίνακας προσπελαύνεται όπως είναι αποθηκευμένος → καλύτερη τοπικότητα αναφορών, αφού γειτονικά στοιχεία προσπελαύνονται σε διαδοχικές επαναλήψεις του εσωτερικού loop

misses συμβαίνουν όταν αναφερόμαστε στο πρώτο στοιχείο κάθε γραμμής (i=0) → συνολικά έχουμε 100 misses

Άσκηση 2

Θεωρούμε το ακόλουθο κομμάτι κώδικα:

```
#define N 1024
float A[N], B[N];
        for(i=0; i<N; i+=1)
                B[i] += 2*A[i];
```

Κάνουμε τις εξής υποθέσεις:

- *Το πρόγραμμα εκτελείται σε έναν επεξεργαστή με μόνο ένα επίπεδο κρυφής μνήμης δεδομένων, η οποία αρχικά είναι άδεια. Η κρυφή μνήμη είναι direct mapped, write-allocate, και έχει μέγεθος 2KB. Το μέγεθος του block είναι 16 bytes.*
- *Το μέγεθος ενός float είναι 4 bytes.*
- *Δήλωση διαδοχικών μεταβλητών (βαθμωτών και μη) στο πρόγραμμα συνεπάγεται αποθήκευσή τους σε διαδοχικές θέσεις στη μνήμη.*

Βρείτε το συνολικό ποσοστό αστοχίας (miss rate) για τις αναφορές που γίνονται στην μνήμη στον παραπάνω κώδικα

```
#define N 1024
float A[N], B[N];
for(i=0; i<N; i+=1)
        B[i] += 2*A[i];
```

- #blocks= 2048/16 = 128
- σε κάθε block => 16/4 = 4 στοιχεία ενός πίνακα
- *Πού απεικονίζονται τα A[i], B[i] ;* στο ίδιο block... *γιατί;*
  - Άρα: read B[0] (m), read A[0] (m), write B[0] (m)

    read B[1] (h) *γιατί;*, read A[1] (m), write B[1] (m)

    read B[2] (h), read A[2] (m), write B[2] (m)

    read B[3] (h), read A[3] (m), write B[3] (m)

    read B[4] (m), read A[4] (m), write B[4] (m)

    ...
- Ανά 4 επαναλήψεις: 9 misses, 3 hits => miss rate = 9/12 = 75%

*Ποιες από τις παρακάτω τεχνικές βελτιστοποίησης **επιπέδου λογισμικού** θα ακολουθούσατε προκειμένου να βελτιώσετε την απόδοση του παραπάνω κώδικα;*

- Loop unrolling

- Merging arrays

- Loop blocking

- Loop distribution

- Loop unrolling

```
for(i=0; i<N; i+=1) {
        B[i] += 2*A[i];
            B[i+1] += 2*A[i+1];
            B[i+2] += 2*A[i+2];
            B[i+3] += 2*A[i+3];
        }
```

- Βοηθάει;
  - *Πού στοχεύει η τεχνική αυτή;*

- Loop blocking

```
for(i=0; i<N; i+=bs)
   for(ii=i; ii<min(i+bs,N); ii++)
        B[ii] += 2*A[ii];
```

- Βοηθάει;
  - *Πού στοχεύει η τεχνική αυτή;*
  - *Ποιο το πρόβλημα απόδοσης του αρχικού κώδικα όσον αφορά τα misses;*

- Loop distribution

```
for(i=0; i<N; i+=1) {
        B[i] += 2*A[i];          =>    ???
}
```

- *Σε τι στοχεύει η τεχνική αυτή;*
- *Υπό ποιες προϋποθέσεις θα βοηθούσε;*

- Merging arrays

```
float A[N], B[N];
for(i=0; i<N; i+=1)
    B[i] += 2*A[i];
```

ο κώδικας γίνεται:

```
struct merge {
    float a;
    float b; };
struct merge merge_array[1024];
for(i=0; i<N; i+=1)
    merge_array[i].b += 2*merge_array[i].a
```

- Σε κάθε block τώρα έχουμε 2 στοιχεία του A και 2 του B
  - Για ζυγά i: 1 miss σε 3 accesses
  - Για μονά i: κανένα miss σε 3 accesses
- Άρα miss rate = 1/6 = 16.67%

*Ποιες από τις παρακάτω τεχνικές βελτιστοποίησης **επιπέδου υλικού** θα ακολουθούσατε προκειμένου να βελτιώσετε την απόδοση του κώδικα;*

- αύξηση block size σε 32 bytes (με διατήρηση της χωρητικότητας)

- αύξηση associativity σε 2-way (με διατήρηση της χωρητικότητας της cache)

- προσθήκη victim cache

- χρήση μηχανισμού hardware prefetching

Ποιο το πρόβλημα απόδοσης του αρχικού κώδικα όσον αφορά τα misses, και πώς μπορεί να βοηθήσει η κάθε τεχνική;

Άσκηση 3

Εξετάζουμε την εκτέλεση του ακόλουθου βρόχου (αντιμετάθεση πίνακα):

```
for(i=0; i<256; i++)
        for(j=0; j<256; j++)
                b[i][j] = a[j][i];
```

- *στοιχεία κινητής υποδιαστολής διπλής ακρίβειας (8 bytes)*
- *ένα επίπεδο data cache: fully associative, write-allocate, 16 KB, LRU πολιτική αντικατάστασης*
- *block size = 64 bytes*
- *οι πίνακες είναι αποθηκευμένοι κατά γραμμές, και "ευθυγραμμισμένοι" ώστε το πρώτο στοιχείο τους να απεικονίζεται στην αρχή μιας γραμμής της cache*

Βρείτε το συνολικό miss rate.

```
for(i=0; i<256; i++)
        for(j=0; j<256; j++)
                b[i][j] = a[j][i];
```

- cache line 64 bytes => 8 στοιχεία πίνακα σε 1 cache line
- Για την αποθήκευση 1 γραμμής του πίνακα => 8*256=2048 bytes, ή 32 cache lines
- Για την αποθήκευση 1 στήλης => 64*256 bytes (*γιατί;*) = 16KB ή 256 cache lines
  - Τα στοιχεία μιας στήλης δεν μπορούν να επαναχρησιμοποιηθούν

```
for(i=0; i<256; i++)
        for(j=0; j<256; j++)
                b[i][j] = a[j][i];
```

Για την 1η επανάληψη του εξωτερικού loop:

a0,0 b0,0 m m

a1,0 b0,1 m h

a2,0 b0,2 m h

...

a7,0 b0,7 m h

a8,0 b0,8 m m (νέα cache line για τον b)

a9,0 b0,9 m h

...

a15,0 b0,15 m h

a16,0 b0,16 m m

...

Το miss pattern επαναλαμβάνεται ανά 8 επαναλήψεις του εσωτερικού loop

- Άρα συνολικά θα έχουμε 256/8 * 9 misses = 288 για 1 επανάληψη του εξ. loop

```
for(i=0; i<256; i++)
        for(j=0; j<256; j++)
                b[i][j] = a[j][i];
```

Για την 1η επανάληψη του εξωτερικού loop:

a0,0 b0,0 m m

a1,0 b0,1 m h

a2,0 b0,2 m h

...

a7,0 b0,7 m h

a8,0 b0,8 m m (νέα cache line για τον b)

a9,0 b0,9 m h

...

a15,0 b0,15 m h

a16,0 b0,16 m m

...

Στην επόμενη επανάληψη του εξ. loop δεν υπάρχει επαναχρησιμοποίηση για κανέναν από τους 2 πίνακες:

- ο b διατρέχεται ούτως ή άλλως κατά γραμμές...
- ο a;

```
for(i=0; i<256; i++)
        for(j=0; j<256; j++)
                b[i][j] = a[j][i];
```

Για την 1η επανάληψη του εξωτερικού loop:

a0,0 b0,0 m m

a1,0 b0,1 m h

a2,0 b0,2 m h

...

a7,0 b0,7 m h

a8,0 b0,8 m m (νέα cache line για τον b)

a9,0 b0,9 m h

...

a15,0 b0,15 m h

a16,0 b0,16 m m

Συμπέρασμα:

- Το miss pattern είναι το ίδιο για κάθε επανάληψη του εξωτερικού loop
- Misses = 256*288 = 73728 , σε σύνολο 2*256*256 αναφορών => miss rate = 56.25%

Εφαρμόστε την τεχνική του blocking στον παραπάνω κώδικα παρουσιάζοντας τον βελτιστοποιημένο κώδικα. Ποιο block size θα επιλέγατε ως καταλληλότερο και γιατί;

```
for(i=0; i<256; i++)
    for(j=0; j<256; j++)
        b[i][j] = a[j][i];
```

=>

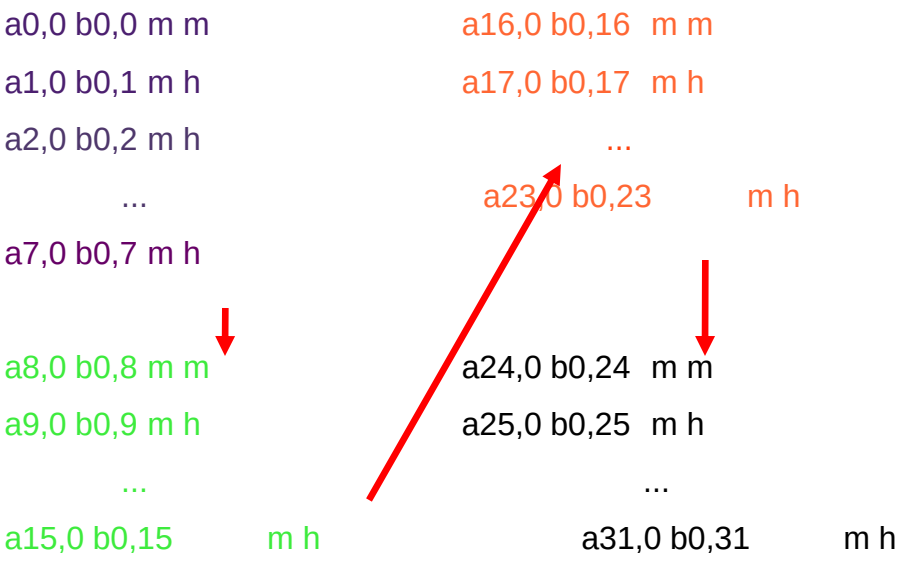
```
for(i=0; i<256; i+=bs)
    for(j=0; j<256; j+=bs)
        for(ii=i; ii<i+bs; ii++)
            for(jj=j; jj<j+bs; jj++)
                b[ii][jj] = a[jj][ii];
```

- Ποιο block size;
  - Τα 2 blocks πρέπει να χωράνε στην cache => κάθε block 8KB ή 1024 στοιχεία = 32x32 στοιχεία => bs =32

Ποιο το ποσοστό αστοχίας για τον blocked κώδικα;

```
for(i=0; i<256; i+=32)
    for(j=0; j<256; j+=32)
        for(ii=i; ii<i+32; ii++)
            for(jj=j; jj<j+32; jj++)
                b[ii][jj] = a[jj][ii];
```

Για την 1η επανάληψη του loop “ii”:

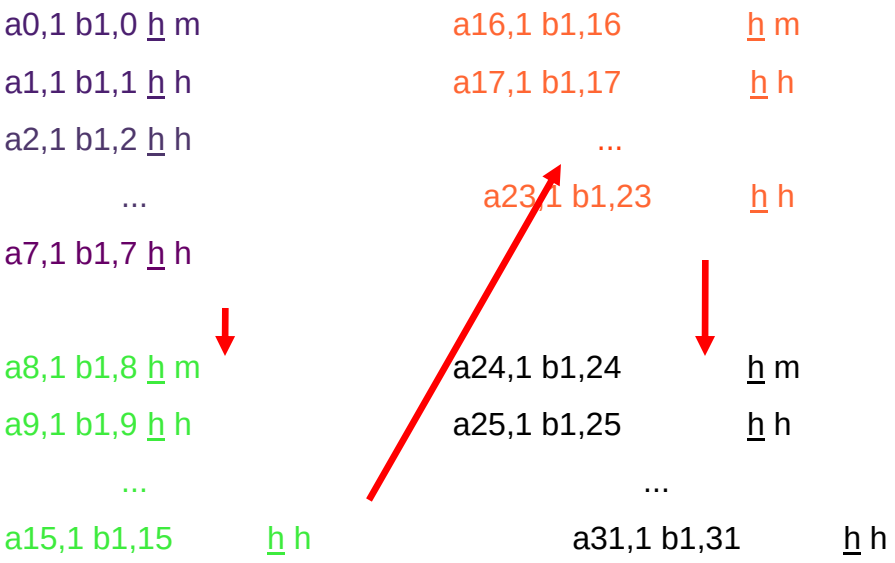


- ανά 8 επαναλήψεις του “jj” το miss pattern επαναλαμβάνεται => 4*9 = 36 misses συνολικά

Ποιο το ποσοστό αστοχίας για τον blocked κώδικα;

```
for(i=0; i<256; i+=32)
    for(j=0; j<256; j+=32)
        for(ii=i; ii<i+32; ii++)
            for(jj=j; jj<j+32; jj++)
                b[ii][jj] = a[jj][ii];
```

Στην 2η επανάληψη του loop “ii”, θα έχουμε επαναχρησιμοποίηση στα στοιχεία του a:



- Σε αυτήν την περίπτωση έχουμε συνολικά 4*1=4 misses

Ποιο το ποσοστό αστοχίας για τον blocked κώδικα;

```
for(i=0; i<256; i+=32)
    for(j=0; j<256; j+=32)
        for(ii=i; ii<i+32; ii++)
            for(jj=j; jj<j+32; jj++)
                b[ii][jj] = a[jj][ii];
```

Η επαναχρησιμοποίηση στα στοιχεία του a θα συνεχιστεί για ii=2,3,...7 (*γιατί;*)

a0,1 b1,0 h m
a1,1 b1,1 h h
a2,1 b1,2 h h
...
a7,1 b1,7 h h

a8,1 b1,8 h m
a9,1 b1,9 h h
...
a15,1 b1,15 h h

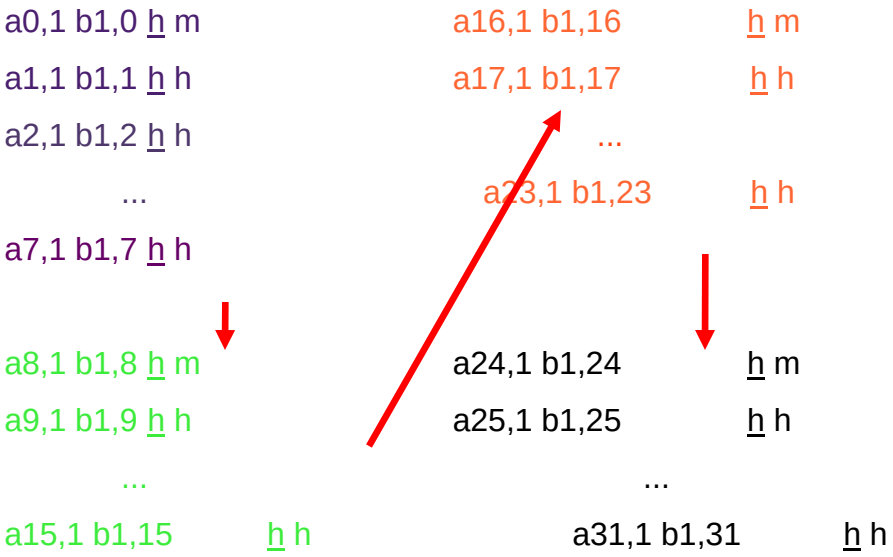
a16,1 b1,16 h m
a17,1 b1,17 h h
...
a23,1 b1,23 h h

a24,1 b1,24 h m
a25,1 b1,25 h h
...
a31,1 b1,31 h h

- Συνολικά, για ii=0,...,7 θα έχουμε 36+7*4=64 misses
- ...το pattern αυτό θα επαναλαμβάνεται για ii=8...15, ii=16...23, ii=24...31 (όπου αλλάζει η στήλη για τον a ώστε να φορτώνεται καινούρια cache line)

Ποιο το ποσοστό αστοχίας για τον blocked κώδικα;

```
for(i=0; i<256; i+=32)
    for(j=0; j<256; j+=32)
        for(ii=i; ii<i+32; ii++)
            for(jj=j; jj<j+32; jj++)
                b[ii][jj] = a[jj][ii];
```



- *Συνολικά επομένως, για μια πλήρη εκτέλεση των 2 εσωτερικότερων loops (για την εκτέλεση δηλαδή του αλγορίθμου σε επίπεδο block πίνακα), θα έχουμε 4*64 = 256 misses σε σύνολο 2*32*32=2048 αναφορών.*
- *Το miss pattern αυτό επαναλαμβάνεται για όλες τις επαναλήψεις των 2 εξωτερικότερων loops, για όλους δηλαδή τους συνδυασμούς blocks των πινάκων a και b (δεν υπάρχει επαναχρησιμοποίηση σε επίπεδο block, μόνο σε επίπεδο στοιχείων εντός του block).*
- *Επομένως, το miss rate = 256/2048 = 12.5%*