ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

ΠΡΟΗΓΜΕΝΑ ΘΕΜΑΤΑ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

8ο εξάμηνο, Ροή Υ, ΗΜΜΥ

4η ΕΡΓΑΣΙΑ

Cache Optimizations

Αλιφιεράκη Ιωάννα Μαρία
ΑΜ: **03107664**

Η/Μ Παράδοσης: **17 Ιουλίου 2011**

Αθήνα,
2011

1 Εισαγωγή

Στην 4η και τελευταία Εργασία θα μελετήσουμε την επίδραση διάφορων τεχνικών βελτιστοποίησης κώδικα -που χρησιμοποιούνται και από διάφορους compilers- και στοχεύουν στην επίτευξη καλύτερης απόδοσης εκτέλεσης κώδικα μέσω καλύτερης αξιοποίησης της Cache.

Για τους σκοπούς της άσκησης θα χρησιμοποιήσουμε τον προσομοιωτή Simics για να προσομοιώσουμε την εκτέλεση ενός προγράμματος σε C που πραγματοποιεί πολλαπλασιασμό δύο τετραγωνικών πινάκων A, B (μεγέθους 256 x 256) με στοιχεία κινητής υποδιαστολής (μεγέθους 8 *bytes*).

1.1 Ιεραρχία Μνήμης

Η ιεραρχία μνήμης που χρησιμοποιήθηκε αποτελείται από **2 επίπεδα cache**.

Η L1 Cache *δεν* είναι ενοποιημένη (unified) και αποτελείται από μία 32 Kb cache εντολών (Instruction Cache) και μία 32 Kb cache δεδομένων (Data Cache) Write-Through με πολιτική LRU. Κάθε τέτοια cache έχει 512 *cache blocks*, μεγέθους 64 *bytes* και είναι 2-way set-associative. Συνοπτικά:

	Size	**Assoc**	**Line number**	**Line size**	**Write policy**	**Replacement policy**
L1I	32K	2	512	64	Write-through	LRU
L1D	32K	2	512	64	Write-through	LRU

Πίνακας 1: Δομή L1 Cache

Η L2 είναι unified, μεγέθους 128 *bytes* με μέγεθος block 128 *bytes* και αποτελείται από 1024 γραμμές. Επιπλέον, είναι Write-back με LRU policy και έχει βαθμό συσχετιστικότητας 4 (4-way set-associative).

	Size	**Assoc**	**Line number**	**Line size**	**Write policy**	**Replacement policy**
L2	128K	4	1024	128	Write-back	LRU

Πίνακας 2: Δομή L2 Cache

Και στις δύο cache έχουμε ορίσει τα read/write miss penalty σε 0 για να περιορίσουμε τον χρόνο προσομοίωσης, όμως στον υπολογισμό των κύκλων εκτέλεσης θα θεωρήσουμε την ποινή στην L1 ίση με *10cc* ενώ στην L2 με *200cc* (βλέπε 1.2 πιο κάτω).

1.2 Μοντέλο Απόδοσης

Το μοντέλο που χρησιμοποιεί ο simics για x86 αρχιτεκτονικές είναι ένας in-order επεξεργαστής με IPC=1. Στην μελέτη μας θεωρούμε πως δεν έχουμε instruction misses και ότι οι εντολές πρόσβασης στην data cache δεν προκαλούν καθυστέρηση εφόσον είναι hit.

Οι ”πραγματικοί” κύκλοι εκτέλεσης δίνονται από τον τύπο:

$$Cycles = Instructions + L1\_misses\ L1\_penalty + L2\_misses\ L2\_penalty \quad (1)$$

2 Τεχνικές Βελτιστοποίησης

2.1 Αρχική έκδοση

Η αρχική και μη βελτιστοποιημένη σε κανέναν βαθμό έκδοση του κώδικα που θα προσομοιώσουμε είναι η αυτή που φαίνεται παρακάτω:

```
for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        for(k=0; k<N; k++)
            C[i][j] += A[i][k]*B[k][j];
```

Τα αποτελέσματα της προσομοίωσης συγκεντώνονται στον παρακάτω πίνακα:

Cycles (in Billion)	0.497
L1 Miss Rate (%)	2.77
L2 Miss Rate (%)	2.12

Πίνακας 3: Αποτέλεσματα για ijk version

2.2 Loop Interchange

Πρώτη τεχνική βελτιστοιποίησης που θα εφαρμόσουμε με στόχο της απόδοσης είναι *η αναδιάταξη βρόχων*. Η τεχνική αυτή στοχεύει στην καλύτερη χρήση της cache μέσω της εκμετάλλευση της χωρικής τοπικότητας των αναφορών σε αυτήν.

Εκτελούμε προσομοιώσεις για όλες τις δυνατές αναδιατάξεις και συγκεντρώνουμε τα αποτελέσματα αυτά στον Πίνακα 2.2.

ijk		
	Cycles (in Billion)	0.497
	L1 Miss Rate (%)	2.77
	L2 Miss Rate (%)	2.12
ikj		
	Cycles (in Billion)	0.489
	L1 Miss Rate (%)	0.46
	L2 Miss Rate (%)	1.23
jki		
	Cycles (in Billion)	1.639
	L1 Miss Rate (%)	8.63
	L2 Miss Rate (%)	9.69

jik		
	Cycles (in Billion)	0.511
	L1 Miss Rate (%)	2.75
	L2 Miss Rate (%)	2.31
kij		
	Cycles (in Billion)	0.507
	L1 Miss Rate (%)	0.48
	L2 Miss Rate (%)	1.41
kji		
	Cycles (in Billion)	1.572
	L1 Miss Rate (%)	8.61
	L2 Miss Rate (%)	9.14

Η αύξηση των miss rates, τόσο στην L1 όσο και στην L2, οδηγούν σε αύξηση και των κύκλων εκτέλεσης. Αυτό είναι αναμενόμενο καθώς, όπως είδαμε και προηγουμένως, το κόστος μιας αστοχίας στην L1 cache "κοστίζει" 10 κύκλους ρολογιού ενώ στην L2 φθάνει τους 200cc. Ο ισχυρισμός αυτός επαληθεύεται και στην πράξη από τα αποτελέσματα του κύκλου προσομοιώσεων που φαίνονται στον παραπάνω πίνακα: όσα προγράμματα είχαν πολλά misses (τάξης μεγέθους περισσότερα - x10) στην L1 και κυρίως στην L2 (πχ jki, kji) είχαν κατ' αναλογία πολύ μεγαλύτερο χρόνο εκτέλεσης.

Είναι προφανές πως οι διαφορετικές παραλλαγές του κώδικα επιτυγχάνουν και διαφορετικές αποδόσεις. Με την έννοια της απόδοσης εννοούμε το χρόνο εκτέλεσης, ο οποίος είδαμε ότι σχετίζεται άμεσα με το ποσοστό αστοχίας των αναφορών στην κρυφή μνήμη. Οι πίνακες αποθηκεύονται στη C

κατά γραμμές στην κύρια μνήμη και όταν υπάρχει κάποιο miss στην κρυφή μεταφέρεται ένα ολόκληρο block στην cache (8 διαδοχικά στοιχεία στην L1 και 16 στην L2). Αναδιατάσσοντας τα indexes των πινάκων ”ζητάμε” διαφορετικά στοιχεία από την cache κάθε φορά τα οποία μπορεί να είναι εκεί (περίπτωση hit) λόγω χωρικής τοπικότητας, ή να μην είναι (περίπτωση miss).

Η σειρά με την οποία γίνονται οι αναφορές στην κρυφή μνήμη καθορίζεται από τον εσωτερικό κώδικα των βρόχων:

```
C[i][j] = C[i][j] + A[i][k] * B[k][j]
```

το οποίο δείχνει πως η σειρά των αναφορών είναι: C[i][j] {r}, A[i][k] {r}, B[k][j] {r}, C[i][j] {w}. Την καλύτερη τοπικότητα αναφορών επιτυγχάνει ο κώδικας με τα **i-k-j** loops. Με αυτό το pattern προσπέλασης ο πίνακας C[**i**][**j**] προσπελαύνεται κατά γραμμή, το ίδιο και ο A[**i**][**k**] και ο B[**k**][**j**]. Το ότι είναι κατά γραμμή φαίνεται από το ότι και στις τρεις περιπτώσεις πρώτα αλλάζει ο δείκτης στήλης και έπειτα ο δείκτης γραμμής. Στα υπόλοιπα patterns η απόδοση μειώνεται ανάλογα με το τι είδους προσπέλαση γίνεται στους πίνακες, δηλαδή σε κάποιες διατάξεις πχ την i-j-k ο C και ο A προσπελαύνονται κατά γραμμή και ο B κατά στήλη. Αυτό έχει σαν αποτέλεσμα να έχουμε περισσότερα misses σε αναφορές στοιχείων του B αλλά και πάλι η απόδοση είναι καλύτερη από την αντίστοιχη j-k-i που όλοι οι πίνακες προσπελαύνονται κατά στήλη.

Το speedup ανά pattern προσπέλασης σε σχέση με τον αρχικό κώδικα δίνεται από τον τύπο:

$$Speedup = \frac{cycles_{ijk}}{cycles_{xxx}} \tag{2}$$

και τα αποτελέσμτα συγκεντρώνονται στον Πίνακα 4.

Pattern	Speedup
ijk	1.00
ikj	1.02
jki	0.30
jik	0.97
kij	0.98
kji	0.32

Πίνακας 4: Speedup per pattern

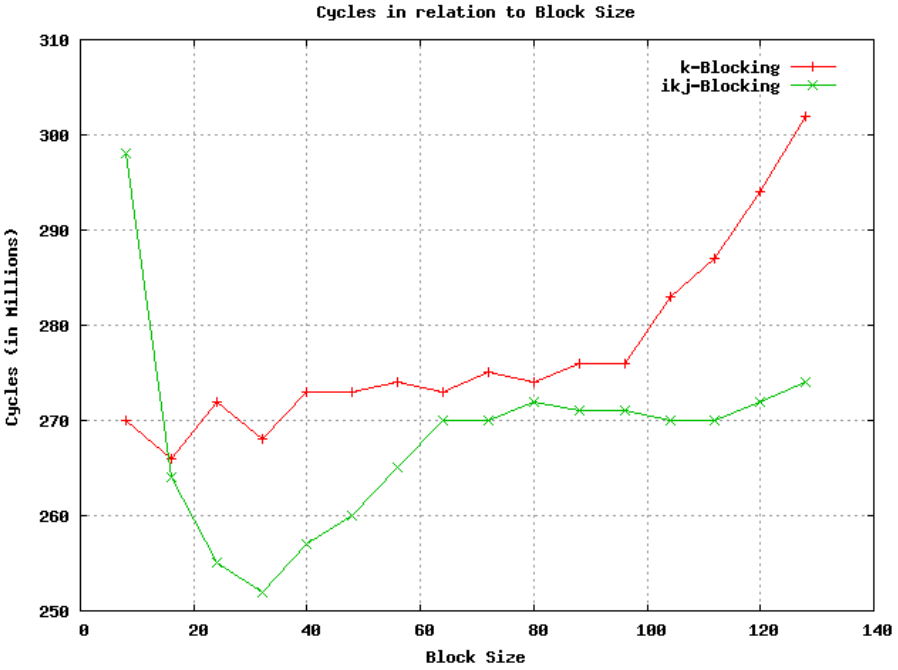
2.3 Cache Blocking

Η επόμενη τεχνική που θα εξετάσουμε είναι αυτή του Blocking (cache). Η γενική ιδέα του cache blocking έγκειται στον διαχωρισμό του χώρου επαναλήψεων ενός loop (loop iteration space) σε μικρότερους υποχώρους, έτσι ώστε το σύνολο δεδομένων (working set) που επεξεργάζεται ο κάθε υποχώρος να χωρά σε κάποιο επίπεδο κρυφής μνήμης, και να μπορεί συνεπώς να επαναχρησιμοποιηθεί εκεί στο μέγιστο δυνατό βαθμό, προτού εκτοπιστεί. Ο διαχωρισμός αυτός γίνεται με βάσει έναν συντελεστή που ονομάζεται blocking size (bs) και επιλέγεται κατάλληλα ώστε η cache να ισομοιράζεται (ει δυνατόν) σε όλες τις δομές δεδομένων που χρησιμοποιεί το πρόγραμμα. Με αυτόν τον τρόπο εκμεταλλευόμαστε την χρονική τοπικότητα των αναφορών στην cache.

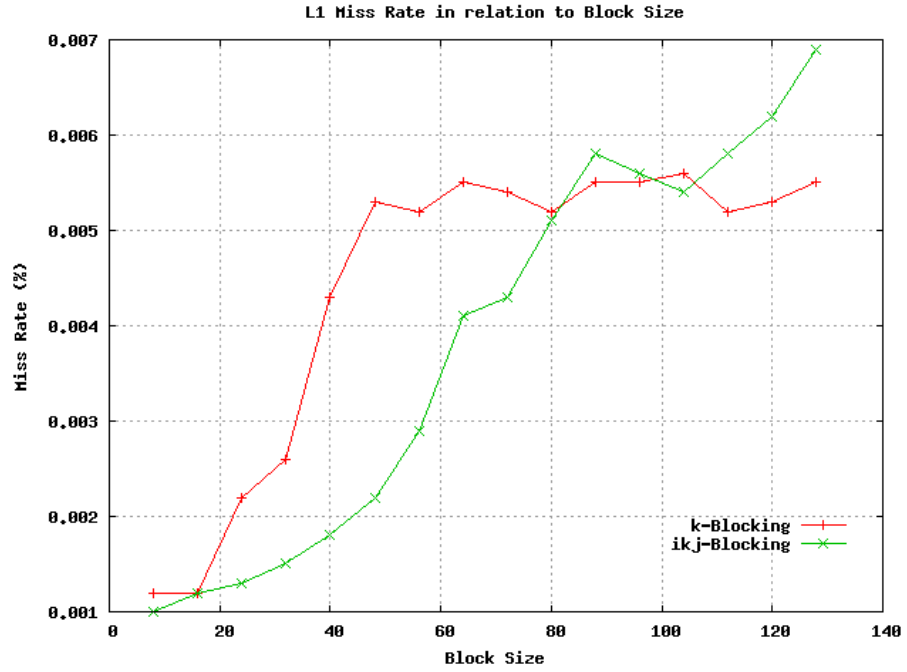
a. Για να εφαρμόσουμε blocking και στα τρία loops του κώδικα πολλαπλασιασμού πινάκων που επιλέξαμε στο Τμήμα 2.2 της εργασίας χρησιμοποιούμε τον παρακάτω κώδικα:

```
for(i=0; i<N; i+=bs)
    for(k=0; k<N; k+=bs)
        for(j=0; j<N; j+=bs)
            for(ii=i; ii<min(i+bs,N); ii++)
                for(kk=k; kk<min(k+bs, N); kk++)
                    for(jj=j; jj<min(j+bs, N); jj++)
                        C[i][j] += A[i][k]*B[k][j];
```

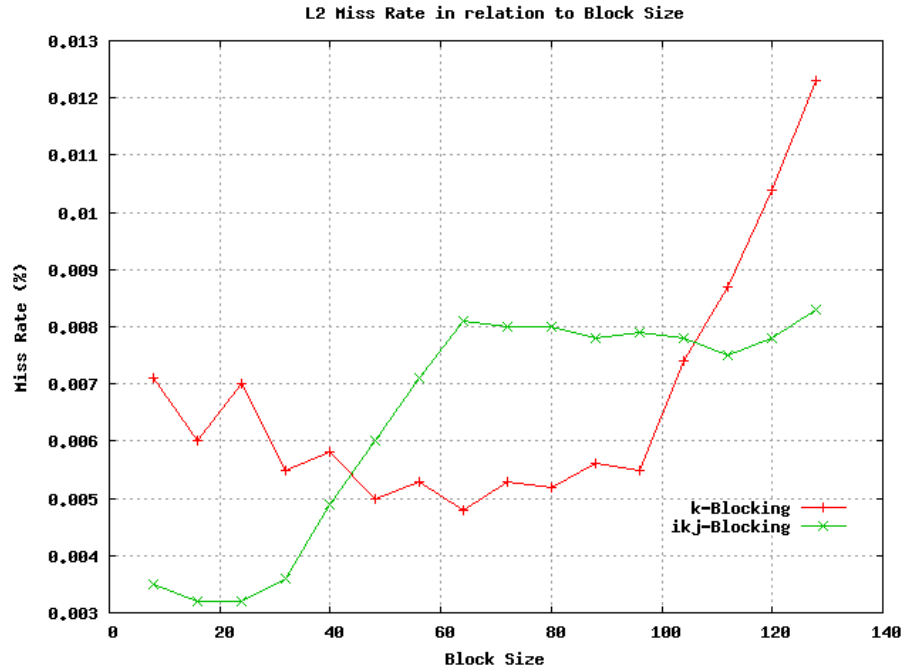
b. Εκτελούμε τώρα μία σειρά προσομοιώσεων των προγραμμάτων των ερωτημάτων a και b με blocking sizes από 8 εώς 128 με βήμα 8 και συγκεντρώνουμε τα αποτελέσματα που αφορούν τον χρόνο εκτέλεσης και τα ποσοστά αστοχίας σε L1 και L2 caches σε διαγράμματα για να μπορούμε να τα αναλύσουμε πιο εύκολα.



Σχήμα 1: Διάγραμμα χρόνου εκτέλεσης συναρτήσει του μεγέθους block



Σχήμα 2: Διάγραμμα ποσοστού αστοχίας στην L1 cache συναρτήσει του μεγέθους block

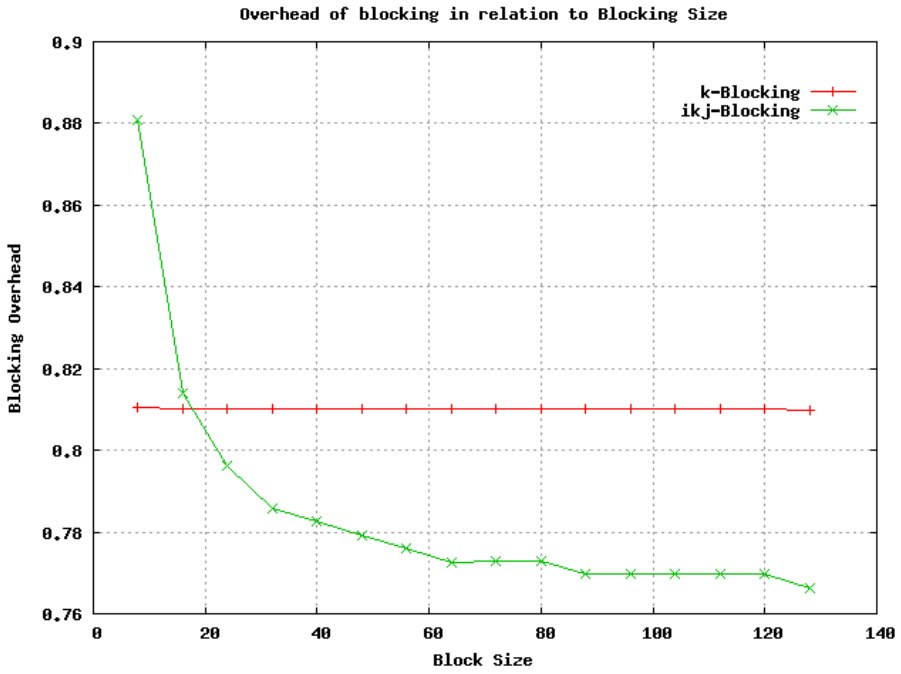


Σχήμα 3: Διάγραμμα ποσοστού αστοχίας στην L2 cache συναρτήσει του μεγέθους block

c. Τώρα θα μελετήσουμε το συνολικό overhead που οφείλεται στην τεχνική του blocking δημιουργώντας διαγράμματα που εκφράζουν την μεταβολή του συνολικού αριθμού εντολών στην περιοχή ενδιαφέροντος (άθροισμα user και supervisor instructions) σε σχέση με τον αριθμό εντολών της interchanged εκδοχής του κώδικα (άξονας y) συναρτήσει του μεταβαλλόμενου blocking size (άξονας x).

Η μεταβολή του αριθμού εντολών της κάθε έκδοσης σε σχέση με τον αντίστοιχο αριθμό της interchanged εκδοχής δίνεται από τον τύπο:

$$\Delta(Instr\_No) = \frac{instr\_executed_{after} \quad instr\_executed_{before}}{clear\_instr\_of\_ikj} \tag{3}$$



Σχήμα 4: Διάγραμμα μεταβολής του συνολικού αριθμού εντολών μιας blocking εκδοχής σε σχέση με την ikj-interchanged συναρτήσει του blocking size

d. Γενικά παρατηρούμε πως το miss rate των blocked εκδόσεων είναι πολύ μικρότερο από τα αντίστοιχα miss rates της αρχικής ή των interchanged εκδοχών. Είναι, επίσης, προφανές πως η επιλογή του blocking factor επηρεάζει τόσο το χρόνο εκτέλεσης όσο και τα miss rates. ’Οπως είπαμε, το blocking size συνήθως επιλέγεται τόσο ώστε να ισοκατανέμεται το μέγεθος της cache στους επιμέρους πίνακες (αποθηκεύονται περίπου ίσοι σε εμβαδό υποπίνακες).

Κατ’ αυτόν τον τρόπο διαφορετικά blocking sizes οδηγούν σε λιγότερα ή περισσότερα misses σε κάθε cache ανάλογα με το μέγεθος αυτής. Γι αυτό και η αύξηση στο blocking size οδηγεί την L1D cache σε περισσότερες αστοχίες ενώ την L2 (ως ένα βαθμό) σε περισσότερες ευστοχίες. Αντίστοιχες μεταβολές παρατηρούνται και στο χρόνο εκτέλεσης των προγραμμάτων αφού, όπως παρατηρήσαμε, είναι άμεσα συνδεδεμένος με τα παραπάνω ποσοστά. Στο διάγραμμα του Σχήματος 1 θα λέγαμε πως συνοψίζεται η επίδραση της μεταβολής του blocking factor σε σχέση με την απόδοση καθώς παρουσιάζει συγκεντρωτικά την επιρροή αυτής και στα δύο επίπεδα της κρυφή μνήμης.

Να σημειώσουμε πως απ' ότι φαίνεται οι καλύτερες επιλογές blocking sizes είναι: **16** για την k-blocking εκδοχή και **32** για την ikj-blocking.

e. Η καλύτερη cache-blocked εκδοχή του κώδικα είναι η **ikj με μέγεθος block 32x32**. Αυτή η εκδοχή οδηγεί σε βελτίωση σε σχέση με την απλοική εκδοχή του κώδικα (Τμήμα 2.1 της εργασίας) που δίνεται από τον Τύπο 2 και είναι: *Speedup=1.972*.

3 Συνολική Εκτίμηση

Συνοψίζοντας, παρατηρούμε πως και οι δύο τεχνικές βελτιστοποίησης οδηγούν σε καλύτερη απόδοση σε σχέση με εκείνη του αρχικού κώδικα.

Η μέγιστη βελτίωση (speedup) που επιτύχαμε με *αναδιάταξη βρόχων* είναι 1.02 ενώ με χρήση του *Cache Blocking* είναι 1.97—τα καταφέρνει σχεδόν δύο φορές καλύτερα!. Επομένως, θα λέγαμε πως ποσοτικά υπερισχύει εμφανώς της πρώτης τεχνικής.

Κρίνοντας, τώρα, ποιοτικά τις δύο τεχνικές θα λέγαμε πως για κώδικα που περιέχει πίνακες (και ειδικά μεγάλους, που δεν χωράν ολόκληροι σε κανένα επίπεδο της κρυφής μνήμης) η ***εκμετάλλευση της χρονικής τοπικότητας*** των αναφορών σε στοιχεία αυτών οδηγεί σε πολύ καλύτερη απόδοση εκτέλεσης σε σχέση με εκμετάλλευση απλώς της χωρικής τοπικότητας αυτών. Την πρώτη βελτίωση, όπως είδαμε, επιτυγχάνουμε με την τεχνική του Cache Blocking (σε όλες τις διαστάσεις των πινάκων) ενώ τη δεύτερη με το Loop Interchange.

4 Παράρτημα

Εφαρμογή του blocking σε όλα τα loops :

```
#include <stdio.h>
#include <stdlib.h>
#define _MAGIC_CASSERT(p) do {                                          \
                typedef int __check_magic_argument[(p) ? 1 :  1];
\
} while (0)
#define MAGIC(n) do {                                   \
                _MAGIC_CASSERT(!(n));                           \
                __asm__ __volatile__ ("xchg %bx,%bx"); \
} while (0)
#define MAGIC_BREAKPOINT MAGIC(0)
inline int min(int a, int b){
        if(a<=b)return a;
        else return b;
}
void init_matrix(float **mat, int n) {
        unsigned int i,j;
        for(i=0; i<n; i++)
             for(j=0; j<n; j++)
                    mat[i][j] = (float)(i+j);
}
int main(int argc, char **argv){
        float **A,**B,**C;
        int i,j,k, ii,jj,kk;
        int N, bf;
        N=atoi(argv[1]);
        bf=atoi(argv[2]);

        A=(float**)malloc(N*sizeof(float*)
```

```
        for(i=0; i<N; i++)
                A[i]=(float*)malloc(N*sizeof(float));
        B=(float**)malloc(N*sizeof(float*));
        for(i=0; i<N; i++)
                B[i]=(float*)malloc(N*sizeof(float));
        C=(float**)malloc(N*sizeof(float*));
        for(i=0; i<N; i++)
                C[i]=(float*)malloc(N*sizeof(float));
        fprintf(stderr, "Initializing matrices...\n");
        init_matrix(A, N);
        init_matrix(B, N);
        init_matrix(C, N);
        MAGIC_BREAKPOINT;
        for(i=0; i<N; i+=bf)
                        for(k=0; k<N; k+=bf)
                for(j=0; j<N; j+=bf)
                        for(ii=i; ii<min(i+bf,N); ii++)
                                for(kk=k; kk<min(k+bf, N); kk++)
                            for(jj=j; jj<min(j+bf, N); jj++)
                                    C[i][j] += A[i][k]*B[k][j];

        MAGIC_BREAKPOINT;
       return 0;
}
```