Θεόδωρος Μπάρκας ΑΜ:2016030050

Προσέγγιση Προβλήματος

Integers Tables

Στο πρόβλημα οριστήκαν 3 συναρτήσεις.
Παραθέτονται παρακάτω και οι 3 για την υλοποίηση με integers .

```
void init2tables(int *A,int *B){
        initTableInt(A);
        initTableInt(B);
}

void initTableInt(int * A){
    for(int i=0;i<arrayWidth;i++){
        for(int j=0;j<arrayWidth;j++){
            A[i*arrayWidth+j]=i*arrayWidth+j;
        }
    }
}

void multiply_tables_3x3(int * A,int *B,int *C){
    for(int i=0;i<arrayWidth;i++){
        for(int j=0;j<arrayWidth;j++){
            int sum=0;
            for(int k=0;k<=arrayWidth;k++){
                sum=sum +A[i*arrayWidth+k]*B[k*arrayWidth+j];   //sum=sum +A[i][k] * B[k][j];
            }
            C[i*arrayWidth+j]=sum;
        }
    }
}
```

Στην υλοποίηση μου επιλέγω να μην χρησιμοποιήσω δισδιάστατους πινάκες άλλα να τους προσομοιώσω σε μοναδιαίο πινάκα μέσω της διαδικασίας A_{ij} = A[i*3+j] δεδομένου ότι i=row και j=column .

Όπως βλέπουμε αρχικά αρχικοποιούμε τις δύο τιμές μέσω της συνάρτησης “ init2Tables ” η οποία καλεί τη συνάρτηση “ initTableInt ” 2 φορές αρχικοποιώντας τους δυο πινάκες ως:

Matrix A

0	1	2
3	4	5
6	7	8

Matrix B

0	1	2
3	4	5
6	7	8

Επομένως το γινόμενο πινάκων που περιμένω να δω στην έξοδο είναι όπως φαίνεται και από το web application αναφοράς είναι ο :

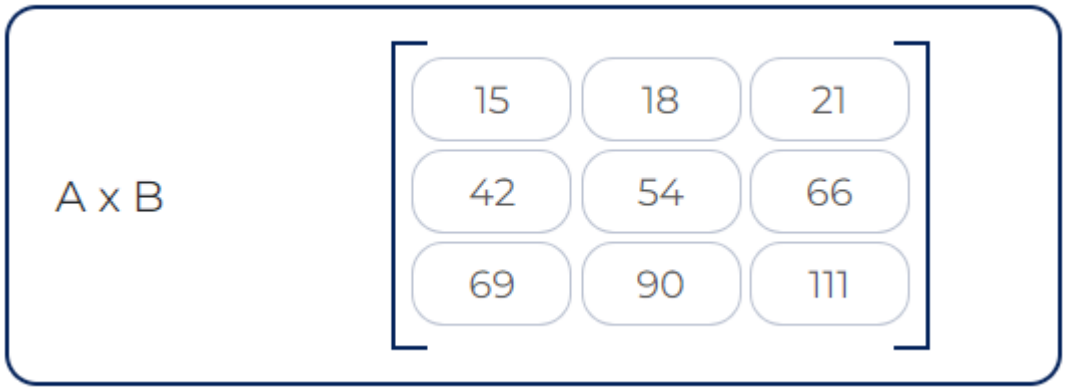

Figure 1: https://ncalculators.com/matrix/3x3-matrix-multiplication-calculator.htm

<u>Σχόλιο υλοποίησης function:“multiply_tables_3x3”</u>
Στην υλοποίηση μου χρησιμοποιώ την βοηθητική μεταβλητή sum αρχικά αρχικοποιημένη στο 0 και μια for για να κάνω τις απαιτούμενες πράξεις μεταξύ της στήλης και της γραμμής όπως περιγράφετε από την θεωρία πολ/σμού πινάκων.

Ύστερα κάλω της απαραίτητες συναρτήσεις από την “main”

```
int main(void)
{
    /*--------------------------------------------------------------------
    Position of arrays stay fixed on this addresses for comparison reason
    It's OK to do this because they have smaller sizes in comparison to
    the float arrays.
    An int contains 2 bytes and a float contains 4 bytes.
    thats why there's no danger of overlapping and overwritten with each other.
    --------------------------------------------------------------------*/
    int * A =(int *) 0x60;
    int * B =(int *) 0x84;
    int * C =(int *) 0xA8;
    init2tables(A,B);
    multiply_tables_3x3(A,B,C);
    while (1)
    {
    }
}
```

Floating Points Tables

Η λογική του προγράμματος παραμένει αναλλοίωτη και για τους floating point .
Το μόνο που αλλάζει είναι ο τύπος των μεταβλητών και ο τύπος των pointers που γίνετε από int → float.

Σύγκριση προσέγγισης με *Floating Point Tables* σε σύγκριση με *Integer Tables(Simulation)*

Χωρικά

Δείξαμε ήδη και φαίνεται και κατά την εκτέλεση του προγράμματος ότι χωρικά καταλαμβάνει περισσότερη μνήμη οι πινάκες floating point σε σύγκριση με τους πινάκες των integers και συγκεκριμένα διπλάσια.

Memory map

0x60-0x83 ή 0x60-0x71	Array A (FLOAT) ή Array A (INT)	Πολλαπλασιαστέος 1
0x84-0xA7 ή 0x84-0x95	Array B (FLOAT) ή Array B (INT)	Πολλαπλασιαστέος 2
0xA8-0xCB ή 0xA8-0xB9	Array C (FLOAT) ή Array C (INT)	Γινόμενο

Βλέπουμε παρακάτω ότι η σχεδίαση του Memory Map συμφωνεί και κατά το simulation:

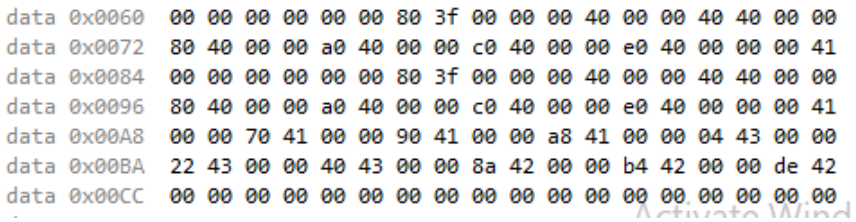

Figure 2: Floating Points Case

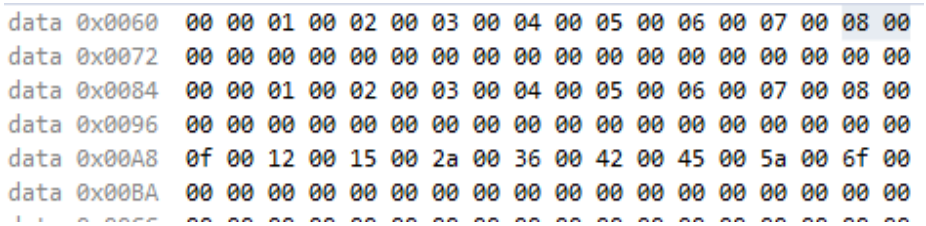

Figure 3: Integers Case

Χρονικά

Ελέγχοντας τον χρόνο εκτέλεσης της συνάρτησης Πολ/σμού και συγκρίνοντας τις τιμές των δυο μεθόδων(int και float) .Διαπιστώσαμε όπως ήταν αναμενόμενο ότι χρονικά ,οι πολλαπλασιασμοί με floating point περνούν παραπάνω χρόνο (Κύκλους ρολογιού).
Αυτό οφείλετε στο μεγαλύτερο πλήθος πράξεων (σε επίπεδο εντολών επεξεργαστή) που εμπεριέχει ένας οποιοσδήποτε πολ/σμός δύο μεταβλητών floating point.
Συγκεκριμένα η υλοποίηση :

- με floating Points παίρνει : 11903-1893=10010 Κύκλους Ρολογιού.
- και αυτή με Integers παίρνει : 2699-573=2126 Κύκλους Ρολογιού.