Σχεδιασμός Ενσωματωμένων Συστημάτων

Αναφορά 1ης Άσκησης | Βάρδια 1 Ομάδα 2

Γρηγόρης Μαντάος	03113171
Γαβαλάς Νίκος	03113121

Platform: Raspberry Pi 3B

GCC Version: gcc Raspbian 4.9.2

Ερώτημα 1

Αρχικά, για τη μέτρηση του χρόνου εκτέλεσης του προγράμματος, προσθέσαμε στο πρόγραμμα την ακόλουθη συνάρτηση βασισμένη στην gettimeofday() που βρίσκεται στην <sys/time.h> :

```
unsigned long cur_time()
{
        struct timeval time;
        gettimeofday(&time, 0);
        return 1000000 * time.tv_sec + time.tv_usec;
}
```

και καλείται ως εξής στην main:

```
unsigned long start = cur_time();

phods_motion_estimation(current, previous, motion_vectors_x,
motion_vectors_y);

unsigned long end = cur_time();

printf("%lu", end - start);
```

Επίσης, για να έχουμε πιο ακριβείς μετρήσεις, για κάθε χρόνο που αναφέρεται παρακάτω στην αναφορά έχουμε χρησιμοποιήσει το εξής script:

measure.py

```
def get_avg(cmd, times):
        s = 0
```

```
        for i in range(times):
                time = int(check_output(cmd).decode())
                s += time
        return int(s / times)

if __name__ == '__main__':
        cmd = [
                './' + sys.argv[1],
                sys.argv[2],
                sys.argv[3]
        ]
        times = int(sys.argv[4])

        print(get_avg(cmd, times))
```

Το οποίο εκτελεί ένα command **times** φορές και υπολογίζει τον μέσο όρο χρόνου εκτέλεσης της phods_motion_estimation από το stdout του προγράμματος.

Έτσι αρχικά υπολογίσαμε τον χρόνο εκτέλεσης του αρχικού script χωρίς καμία αλλαγή:

```
./measure.py phods 16 16 1000
```

Όπου είδαμε ότι το αρχικό πρόγραμα τρέχει σε **80000 - 83000 us**.

Ερώτημα 2

Merge

Το πρώτο που παρατηρήσαμε ήταν ότι το κύριο μέρος των loop είχε αυτή τη μορφή:

```
/*For all candidate blocks in X dimension*/
for(i=-S; i<S+1; i+=S)
{
        distx = 0;

        /*For all pixels in the block*/
        for(k=0; k<Bx; k++)
                for(l=0; l<By; l++)
                        ... // Calculations for X
}
/*For all candidate blocks in Y dimension*/
for(i=-S; i<S+1; i+=S)
{
        disty = 0;

        /*For all pixels in the block*/
        for(k=0; k<Bx; k++)
```

```
                for(l=0; l<By; l++)
                        ... // Calculations for Y
}
```

Οπότε κάναμε **merge** τους παραπάνω κλάδους και το φέραμε στη μορφή:

```
/*For all candidate blocks in both dimensions*/
for(i=-S; i<S+1; i+=S)
{
        distx = 0;
        disty = 0;

        /*For all pixels in the block*/
        for(k=0; k<Bx; k++)
                for(l=0; l<By; l++)
                        ... // Calculations for X
                        ... // Calculations for Y
}
```

Το οποίο έδωσε περίπου **2% βελτίωση** φέρνοντας τον χρόνο εκτέλεσης μόλις κάτω από τα **80000 us**.

Unroll

Ύστερα παρατηρήσαμε ότι μέσα στα loops των x και y, τα S και i δεν κάνουν loop εώς κάποιο όριο που εξαρτάται από το input, αλλά πάνω σε σταθερές τιμές, οπότε χρησιμοποιήσαμε macros για να τα κάνουμε unroll, ως εξής:

```
#define LOOP_I(i)\
        distx = 0;\
        disty = 0;\
        for (k = 0; k < Bx; k++)\
                for (l = 0; l < By; l++)\
                        ... // Calculations for X
                        ... // Calculations for Y

#define LOOP_S\
        min1 = 255 * Bx * Bx;\
        min2 = 255 * By * By;\
        LOOP_I(-S);\
        LOOP_I(0);\
        LOOP_I(S);\
        vectors_x[x][y] += bestx;\
        vectors_y[x][y] += besty;
```

Οπότε το σώμα του κυρίος προγράμματος τώρα έγινε:

```
/*For all blocks in the current frame*/
for (x = 0; x < N / Bx; x++)
{
        for (y = 0; y < M / By; y++)
        {
                LOOP_S(4);
                LOOP_S(2);
                LOOP_S(1);
        }
}
```

Όμως και με αυτόν τον μετασχηματισμό η βελτίωση στον χρόνο εκτέλεσης φαινόταν να ήταν γύρω στα 1-2ms, within the margin of error.

P.S: Χρησιμοποιόντας macros ουστιαστικά κάνουμε το trade-off χρόνου εκτέλεσης για binary size του εκτελέσιμου και compile time.

Data Reuse

Τέλος, παρατηρήσαμε ότι κατα τον έλεγχο των pixel πολές τιμές στα if conditions και τους δείκτες των πινάκων χρησιμοποιόντουσαν πολλές φορές, οπότε εφαρμόσαμε το εξής:

```
int bxxk = Bx * x + k;
int byyl = By * y + l;

p1 = current[bxxk][byyl];
q1 = current[bxxk][byyl];

bxxk += vectors_x[x][y];
byyl += vectors_x[x][y];

if ((bxxk + i) < 0 ||
        (bxxk + i) > (N - 1) ||
        (byyl) < 0 ||
        (byyl) > (M - 1))
{
        p2 = 0;
}
else
{
        p2 = previous[bxxk + i][byyl];\
}
/* Και αντίστοιχα για το Y (βλ. main.c) */
```

Το οποίο έριξε τον χρόνο εκτέλεσης στα **30000 us**, δηλαδή πάνω από **60% βελτίωση**!

Ερώτημα 3

Για τον υπολογίσμο του βέλτιστου block size υλοποιήσαμε το python script:

explore_block_size.py

Σύμφωνα με το οπόιο πήραμε τις παρακάτω μετρήσεις για όλα τα block sizes που είναι κοινοί διαιρέτες του N και του M.

Block Size	Time (us)
1	45369
2	36552
4	31519
8	30508
16	30336

Best:

- Block Size: **16x16**
- Execution Time: **30336**

Ερώτημα 4

Για τον υπολογίσμο του βέλτιστου block size υλοποιήσαμε το python script:

explore_rectangle_size.py

Σύμφωνα με το οπόιο πήραμε τις παρακάτω μετρήσεις για όλους τους διαιρέτες του N και του M.

Block Size	Time (us)	Block Size	Time (us)	Block Size	Time (us)
1x1	46713	8x1	34991	24x1	35253
1x2	38308	8x2	32492	24x2	32028
1x4	33721	8x4	30883	24x4	31700
1x8	31989	8x8	30429	24x8	30635
1x11	31391	8x11	29033	24x11	30019
1x16	30619	8x16	30044	24x16	29736
1x22	29846	8x22	29551	24x22	29811
1x44	29543	8x44	28675	24x44	29199
1x88	29256	8x88	29167	24x88	29656
1x176	29492	8x176	28704	24x176	30000
2x1	41202	9x1	35443	36x1	33919

2x2	36654	9x2	31716	36x2	32226
2x4	33134	9x4	31639	36x4	30736
2x8	30589	9x8	30558	36x8	29609
2x11	31249	9x11	30014	36x11	30574
2x16	30444	9x16	30884	36x16	29868
2x22	30104	9x22	30408	36x22	29256
2x44	29191	9x44	29753	36x44	30047
2x88	28797	9x88	29658	36x88	29281
2x176	29112	9x176	30041	36x176	29295
3x1	38393	12x1	35048	48x1	34003
3x2	35116	12x2	32554	48x2	32541
3x4	32309	12x4	31250	48x4	29989
3x8	30910	12x8	30610	48x8	29237
3x11	30395	12x11	29878	48x11	30915
3x16	30816	12x16	29565	48x16	30175
3x22	30249	12x22	29384	48x22	30346
3x44	29425	12x44	29497	48x44	28292
3x88	29519	12x88	29651	48x88	29403
3x176	28562	12x176	29308	48x176	29186
4x1	37451	16x1	34541	72x1	33648
4x2	34525	16x2	32497	72x2	32503
4x4	31403	16x4	30527	72x4	30994
4x8	30561	16x8	30717	72x8	30477
4x11	30228	16x11	29846	72x11	30867
4x16	29033	16x16	30077	72x16	29851
4x22	29787	16x22	30319	72x22	30023
4x44	29789	16x44	29303	72x44	29619
4x88	28613	16x88	29713	72x88	29033
4x176	30049	16x176	29026	72x176	29424
6x1	36444	18x1	35124	144x1	34715

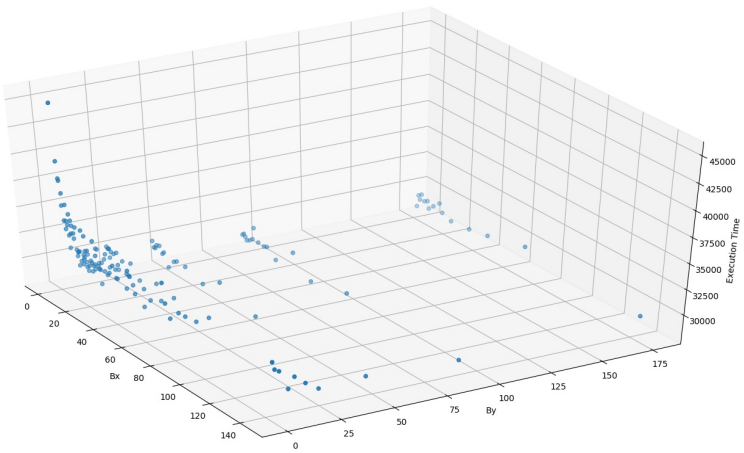
6x2	34143	18x2	32345	144x2	33401
6x4	31396	18x4	31020	144x4	31832
6x8	30220	18x8	29975	144x8	31489
6x11	30374	18x11	30544	144x11	30304
6x16	30428	18x16	30457	144x16	30495
6x22	29480	18x22	29580	144x22	30267
6x44	28924	18x44	28717	144x44	30198
6x88	28816	18x88	29476	144x88	30126
6x176	29647	18x176	30062	144x176	29657

Best:

- Block Size: **48x44**
- Execution Time: **28292**

Κατασκευάζοντας ένα 3D scatterplot για τα άνωθι δεδομένα με χρήση του script:

plot.py



καταλήγουμε στο συμπέρασμα ότι τα σημεία σχηματίζουν μία convex γραφική παράσταση, που παρούσιάζει ένα όλικο ελάχιστο.

Μπορούμε να προσεγγίσουμε αυτό το ελάχιστο σχετικά γρήγορα με διάφορες αριθμητικές μεθόδους, χωρίς να χρειάζεται να αναζητήσουμε εξαντλητικά το Design Space.

Η πιο απλή τέτοια μέθοδος θα ήταν να κάναμε δυαδική αναζήτηση σε καθένα από τα Bx, By βρίσκοντας για ποιο Bx και By αντίστοιχα έχουμε μικρότερη τιμή για τον χρόνο εκτέλεσης.