National Technical University of Athens
Electrical and Computer Engineering

Embedded Systems

1st Lab

OPTIMISATION OF ALGORITHMS FOR LOW ENERGY CONSUMPTION AND HIGH PERFORMANCE

Nanos Georgios
nanosgiwrgos1997@gmail.com

1st Issue : Loop Optimizations & Design Space Exploration

On our personal computer we did the following searches:

1. Using Unix commands, we recorded the characteristics of the computer on which we performed the laboratory exercise. For each feature, we noted the corresponding command with which to find the requested information. Specifically:

- lsb_release -r
- uname -r
- lscpu
- sudo lshw -short
- vi /proc/cpuinfo/

Version of Operating System	Linux Kernel	CPU(s)	CPU (MHz)
Ubuntu 18.04	5.4.0-53-generic	4	800.055

L1d cache	L1i cache	L2 cache	L3 cache	RAM	Block size
32K	32K	256K	3072K	4GB	64

2. We then transformed the given code to measure the time it takes the phods_motion_estimation function to to execute. The time was measured using the gettimeofday function with with microsecond precision. The code after being executed 10 times, we measured the average, maximum and minimum execution time. measurement results are shown below: main(), after transforming it to measure the execution time phods_motion_estimation, is as follows:

```
int main()
{
    double time;
    struct timeval ts,tf;
    int current[N][M], previous[N][M], motion_vectors_x[N/B][M/B],
    motion_vectors_y[N/B][M/B], i, j;
    read_sequence(current,previous);

    gettimeofday(&ts, NULL);
    phods_motion_estimation(current, previous,motion_vectors_x,
   motion_vectors_y);
    gettimeofday(&tf,NULL);
    time=(tf.tv_sec-ts.tv_sec)+(tf.tv_usec-ts.tv_usec) * 0.000001;
    printf("%lf\n", time);
    return 0;
}
```

Listing 1: phods_motion_estimation with gettimeofday function

The python script (time.py) that we used to run ten times the code and print (min, average, max), is as follows:

```
#!/usr/bin/python3
import sys
import time
from subprocess import check_output

```

```
if __name__ == '__main__':
    if len(sys.argv) != 3:
        print("Usage: time.py [phods/phods_opt] <iterations>")
        raise SystemExit
    cmd = ['./' + sys.argv[1]]
    iterations = int(sys.argv[2])

    sum_time = 0
    min_time = 1
    max_time = 0

    for i in range(iterations):
        value = float(check_output(cmd).decode())
        #print('Value', i, ':', value)
        sum_time += value
        if value < min_time:
        min_time = value
        if value > max_time:
        max_time = value
    avg_time = sum_time/iterations
    print('(min, average, max)= ', min_time, max_time, str.format('{0:.6f}',
   avg_time))
```

Listing 2: time.py

Minimum (s)	Average (s)	Maximum (s)
0.006673	0.006882	0.007483

3. Given the existence of the infrastructure for measuring time it takes for the critical part of the application to run, we applied transformations to the code to reduce the time it takes execution time. For each transformation, we recorded the corresponding change and justify why we did it. After completed the transformations in the code, we re-executed the code 10 times and measured the average, maximum and minimum time run time, respectively, as in question 2. Finally, we compared the times execution times with those of query 2. Initially, it is observed that the tracks that distx and disty are in execution loops with identical edges and are independent of each other each other. Therefore we merged them (loop fusion) into a common loop.

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

We re-run the script in the updated phods.c (phods_v2.c) and we have times:

Minimum (s)	Average (s)	Maximum (s)
0.006320	0.006508	0.006815

We observe a slight improvement in runtime of 5.4 Next, we observe that within the loops of x and y, S and i do not loop up to some input-dependent threshold, but over fixed values, so we used macros to unroll them. (Loop Unrolling). The code (phods_v3.c) after these changes is as follows:

```
for(x=0; x<N/B; x++)
{
    for(y=0; y<M/B; y++)
    {
        FOR_LOOP_S(4);
        FOR_LOOP_S(2);
        FOR_LOOP_S(1);
    }
}
```

where,

```
#define FOR_LOOP_S(s) /
    min1 = 255*B*B; /
    min2 = 255*B*B; /
    FOR_LOOP_I(-s); /
    FOR_LOOP_I(0); /
    FOR_LOOP_I(s); /
    vectors_x[x][y] += bestx; /
    vectors_y[x][y] += besty;
```

and

```
#define FOR_LOOP_I(i) /
    distx = 0; /
    disty = 0; /
    for(k=0;k<B;k++) /
    { /
        [...] /
    } /
    if(distx<min1) /
    { /
        min1= distx; /
        bestx = i; /
    } /
    if(disty<min2) /
    { /
        min2=disty; /
        besty=i;
    }
```

We re-run the script in the updated phods.c (phods_v3.c) and we have times:

Minimum (s)	Average (s)	Maximum (s)
0.006020	0.006175	0.006421

We again observe a slight improvement in execution time compared to the phods_v2 of 5.1Finally, we observed how the above macros are accessed the elements of the matrix vectors_x[x][y] and vectors_y[x][y] multiple times within each iteration, without them having changed value. Therefore, the code has been modified to compute these elements only once once in each iteration of the overall double loop. Thus, when is used again, the external memory is not accessed again, because the value is in the processor's cache memory. The new updated code phods_v4.c is as follows:

```
for(x=0; x<N/B; x++)
{
    for(y=0; y<M/B; y++)
    {
        int array_x = vectors_x[x][y];
        int array_y = vectors_y[x][y];
        FOR_LOOP_S(3);
```

```
        FOR_LOOP_S(2);
        FOR_LOOP_S(1);
    }
}
```

In addition, we have obviously substituted in the FOR_LOOP_I macros and FOR_LOOP_S the vectors_x[x][y] with array_x and vectors_y[x][y] with array_y. The resulting new times are as follows:

Minimum (s)	Average (s)	Maximum (s)
0.003085	0.003243	0.003555

This time we see a much greater improvement in time runtime compared to phods_v3, which is of the order of 47.5In conclusion, in terms of overall time improvement, thanks to loop fusion, loop unrolling and data reuse, from the original phods, to final phods_v4, the average has been reduced by 52.9%. Obviously, as well as we have seen from the previous results, data reuse played the biggest role, since the memory access instructions, as we know, are the ones that the most overhead in the execution of a program.

4. In the code resulting from question 3 (phods_v4), we applied Design Space Exploration with respect to finding the optimal block size B (variable in the code) for the architecture of our computer architecture. The search was performed by exhaustive search and only for values of B that are common divisors of M and N. We have that M = 176 and N = 144, so the common divisors are [1, 2, 4, 8, 16]. Therefore, for each of the 5 possible values of B, the program was executed 10 times and the performance metric was defined as the average of The performance metric was defined as the runtime over these 10 runs. We note that phods.c has itself set the Block Size value to 16 (#define B 16), Therefore, it was necessary to modify the code to accept as parameter B. Specifically, we added the following lines to the beginning of main().

```
if (argc == 1)
{
    B = 16;
}
else if (argc == 2)
{
    B = atoi(argv[1]);
}
else
{
    printf("phods_v4 usage: ./phods_v4 block_size\n");
    exit(1);
}
```

The time_block.py script we ran to show us the averages times for all values of B is the following:

```
#!/usr/bin/python3
import sys
from subprocess import check_output
if __name__ == '__main__':
    if len(sys.argv) != 3:
         print("Usage: python3 time_block.py phods_v4 [iterations]")
         raise SystemExit
    if sys.argv[1] == 'phods_v4':
        block_sizes = [1, 2, 4, 8, 16]
        for N in block_sizes:
            cmd = ['./' + sys.argv[1], str(N)]
            iterations = int(sys.argv[2])
            sum_time = 0
```

```
            for i in range(iterations):
                value = float(check_output(cmd).decode())
                sum_time += value
            avg_time = suma/iterations
            print('Block size', N, ':', str.format('{0:.6f}', avg_time), 'sec
    ')
     else:
         print("Usage: python3 time_block.py phods_v4 [iterations]")
         raise SystemExit
```

Block Size	Average Time
1	0.005719
2	0.004396
4	0.003655
8	0.003595
16	0.003218

So, the value of B for which the best return is obtained is 16. Finally, we observed that there is some relationship between this value and the block size of the cache we recorded in question 1. More specifically, as we saw from the phods_motion_estimation algorithm the images (current - previous) are divided into blocks of size BxB to compared with each other. Each such block is transferred to the cache and if it fits, it takes up a certain amount of space, as much as the cache block size. The smaller the B, the more blocks to be allocated in the cache and therefore the more memory accesses we will have. The optimum would obviously result in the case where B would be equal to the cache block size, i.e. 64. (This would cover the least number of blocks in the cache).
5. In the code resulting from question 3, we applied Design Space Exploration with respect to finding the optimal block size considering a rectangular block of dimensions Bx and By for its architecture The size of Bx to be a divisor of N and sector size By be a divisor of M.
N=144 so, Bx = [1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 36, 48, 72, 144]
M=176 so, By = [1, 2, 4, 8, 11, 16, 22, 44, 88, 176]
We applied exhaustive search in order to find the optimal pair Bx, By which minimizes the execution time of the program. For each pair Bx and By, the program was executed 10 times and as a metric performance metric was defined as the average of the execution time in these 10 executions. Obviously, the code was adapted to accept now 2 different dimensions for Block. This is in the file phods_v4_2d.c. The changes we had to make to main() as well are as follows:

```
if (argc == 1)
{
    Bx = 16;
    By = 16;
}
else if (argc == 3)
{
    Bx = atoi(argv[1]);
    By = atoi(argv[1]);
}
else
{
    printf("Usage: ./phods_v4_2d block_size_x block_size_y\n");
    exit(1);
}
```

We again created a script (time_block2) which is as follows:

```
#!/usr/bin/python3
```

```
import sys
from subprocess import check_output
if __name__ == '__main__':
    if len(sys.argv) != 3:
    print("Usage: python3 time_block2.py phods_v4_2d [iterations]")
    raise SystemExit
    if sys.argv[1] == 'phods_v4_2d':
        N = 144
        M = 176
        minimum = 1
        bx = 1
        by = 1

        for bx in range (1,N+1,1):
            if N%bx == 0:
            for by in range (1,M+1,1):
                if M%by == 0:
                    cmd = ['./' + sys.argv[1], str(bx), str(by)]
                    iterations = int(sys.argv[2])
                    sum_time = 0
                    for i in range(iterations):
                    value = float(check_output(cmd).decode())
                    sum_time += value
                    avg_time = sum_time/iterations
                    print('Block size', bx,'x',by,':',
                    str.format('{0:.6f}', avg_time), 'sec')
                if avg_time < minimum:
                    minimum = avg_time
                    bx_min = bx
                    by_min = by
        print('Best block size:', bx_min, 'x', by_min, ':', str.format('{0:.6
    f}', minimum), 'sec')
     else:
        print("Usage: python3 time_block2.py phods_v4_2d [iterations]")
        raise SystemExit
```

Results from time_block2:

```
Block size 1 x 1 : 0.005855 sec
Block size 1 x 2 : 0.005895 sec
Block size 1 x 4 : 0.006029 sec
Block size 1 x 8 : 0.005884 sec
Block size 1 x 11 : 0.005823 sec
Block size 1 x 16 : 0.006032 sec
Block size 1 x 22 : 0.005984 sec
Block size 1 x 44 : 0.005939 sec
Block size 1 x 88 : 0.005894 sec
Block size 1 x 176 : 0.006261 sec
Block size 2 x 1 : 0.004408 sec
Block size 2 x 2 : 0.004447 sec
Block size 2 x 4 : 0.004424 sec
Block size 2 x 8 : 0.004433 sec
Block size 2 x 11 : 0.004562 sec
Block size 2 x 16 : 0.004453 sec
Block size 2 x 22 : 0.004368 sec
Block size 2 x 44 : 0.004405 sec
Block size 2 x 88 : 0.004740 sec
Block size 2 x 176 : 0.004528 sec
Block size 3 x 1 : 0.003949 sec
Block size 3 x 2 : 0.003894 sec
Block size 3 x 4 : 0.004158 sec
Block size 3 x 8 : 0.004018 sec
```

```
Block size 3 x 11 : 0.003973 sec
Block size 3 x 16 : 0.003922 sec
Block size 3 x 22 : 0.003879 sec
Block size 3 x 44 : 0.004064 sec
Block size 3 x 88 : 0.003918 sec
Block size 3 x 176 : 0.003901 sec
Block size 4 x 1 : 0.003720 sec
Block size 4 x 2 : 0.003585 sec
Block size 4 x 4 : 0.003992 sec
Block size 4 x 8 : 0.003743 sec
Block size 4 x 11 : 0.003660 sec
Block size 4 x 16 : 0.003670 sec
Block size 4 x 22 : 0.003675 sec
Block size 4 x 44 : 0.004166 sec
Block size 4 x 88 : 0.003650 sec
Block size 4 x 176 : 0.003604 sec
Block size 6 x 1 : 0.003500 sec
Block size 6 x 2 : 0.003477 sec
Block size 6 x 4 : 0.003614 sec
Block size 6 x 8 : 0.003552 sec
Block size 6 x 11 : 0.003512 sec
Block size 6 x 16 : 0.003485 sec
Block size 6 x 22 : 0.003421 sec
Block size 6 x 44 : 0.003357 sec
Block size 6 x 88 : 0.003642 sec
Block size 6 x 176 : 0.003489 sec
Block size 8 x 1 : 0.003392 sec
Block size 8 x 2 : 0.003321 sec
Block size 8 x 4 : 0.003489 sec
Block size 8 x 8 : 0.003718 sec
Block size 8 x 11 : 0.003450 sec
Block size 8 x 16 : 0.003483 sec
Block size 8 x 22 : 0.003471 sec
Block size 8 x 44 : 0.003751 sec
Block size 8 x 88 : 0.003644 sec
Block size 8 x 176 : 0.003467 sec
Block size 9 x 1 : 0.003341 sec
Block size 9 x 2 : 0.003239 sec
Block size 9 x 4 : 0.003302 sec
Block size 9 x 8 : 0.003262 sec
Block size 9 x 11 : 0.003508 sec
Block size 9 x 16 : 0.003261 sec
Block size 9 x 22 : 0.003247 sec
Block size 9 x 44 : 0.003193 sec
Block size 9 x 88 : 0.003403 sec
Block size 9 x 176 : 0.003458 sec
Block size 12 x 1 : 0.003224 sec
Block size 12 x 2 : 0.003129 sec
Block size 12 x 4 : 0.003198 sec
Block size 12 x 8 : 0.003191 sec
Block size 12 x 11 : 0.003227 sec
Block size 12 x 16 : 0.003326 sec
Block size 12 x 22 : 0.003188 sec
Block size 12 x 44 : 0.003236 sec
Block size 12 x 88 : 0.003261 sec
Block size 12 x 176 : 0.003113 sec
Block size 16 x 1 : 0.003336 sec
Block size 16 x 2 : 0.003414 sec
Block size 16 x 4 : 0.003404 sec
Block size 16 x 8 : 0.003200 sec
Block size 16 x 11 : 0.003347 sec
```

```
Block size 16 x 16 : 0.003324 sec
Block size 16 x 22 : 0.003482 sec
Block size 16 x 44 : 0.003280 sec
Block size 16 x 88 : 0.003210 sec
Block size 16 x 176 : 0.003266 sec
Block size 18 x 1 : 0.003038 sec
Block size 18 x 2 : 0.003071 sec
Block size 18 x 4 : 0.003066 sec
Block size 18 x 8 : 0.002949 sec
Block size 18 x 11 : 0.002988 sec
Block size 18 x 16 : 0.003080 sec
Block size 18 x 22 : 0.003079 sec
Block size 18 x 44 : 0.003041 sec
Block size 18 x 88 : 0.003126 sec
Block size 18 x 176 : 0.003021 sec
Block size 24 x 1 : 0.003089 sec
Block size 24 x 2 : 0.003076 sec
Block size 24 x 4 : 0.003164 sec
Block size 24 x 8 : 0.003335 sec
Block size 24 x 11 : 0.003063 sec
Block size 24 x 16 : 0.003141 sec
Block size 24 x 22 : 0.003102 sec
Block size 24 x 44 : 0.003068 sec
Block size 24 x 88 : 0.003111 sec
Block size 24 x 176 : 0.003260 sec
Block size 36 x 1 : 0.002734 sec
Block size 36 x 2 : 0.002736 sec
Block size 36 x 4 : 0.002671 sec
Block size 36 x 8 : 0.002649 sec
Block size 36 x 11 : 0.002716 sec
Block size 36 x 16 : 0.002996 sec
Block size 36 x 22 : 0.002673 sec
Block size 36 x 44 : 0.002688 sec
Block size 36 x 88 : 0.002638 sec
Block size 36 x 176 : 0.002755 sec
Block size 48 x 1 : 0.002686 sec
Block size 48 x 2 : 0.002730 sec
Block size 48 x 4 : 0.002913 sec
Block size 48 x 8 : 0.002736 sec
Block size 48 x 11 : 0.002738 sec
Block size 48 x 16 : 0.002767 sec
Block size 48 x 22 : 0.002713 sec
Block size 48 x 44 : 0.002667 sec
Block size 48 x 88 : 0.002777 sec
Block size 48 x 176 : 0.002729 sec
Block size 72 x 1 : 0.002758 sec
Block size 72 x 2 : 0.002687 sec
Block size 72 x 4 : 0.002690 sec
Block size 72 x 8 : 0.002728 sec
Block size 72 x 11 : 0.002770 sec
Block size 72 x 16 : 0.002666 sec
Block size 72 x 22 : 0.002796 sec
Block size 72 x 44 : 0.002698 sec
Block size 72 x 88 : 0.002755 sec
Block size 72 x 176 : 0.002734 sec
Block size 144 x 1 : 0.002718 sec
Block size 144 x 2 : 0.002638 sec
Block size 144 x 4 : 0.002622 sec
Block size 144 x 8 : 0.002602 sec
Block size 144 x 11 : 0.002737 sec
Block size 144 x 16 : 0.002887 sec
```

```
Block size 144 x 22 : 0.002660 sec
Block size 144 x 44 : 0.002740 sec
Block size 144 x 88 : 0.002631 sec
Block size 144 x 176 : 0.002689 sec
```

We recorded the pair Bx, By for which we obtained the best performance:

```
Best block size: 144 x 8 : 0.002602 sec
```

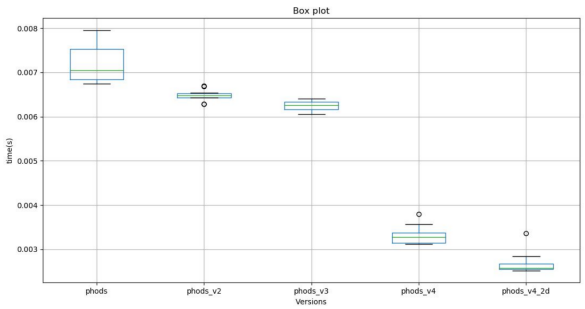
A curious way of limiting the search: Based on the previous question, B=16 was the optimal choice for the shortest runtime. Therefore, the search could be limited to blocks that are greater than or equal to 16x16.

6.Finally, we visualized the results of questions 2-5 in a boxplot and compared-analyzed the experimental results obtained. We used the following box_plots.py script.

```
#!/usr/bin/python3
import sys
import time
from subprocess import check_output
import pandas as pd
import matplotlib.pyplot as plt
if __name__ == '__main__':
    iterations = 10
    phods=[]
    phods_v2=[]
    phods_v3=[]
    phods_v4=[]
    phods_v4_2d=[]
    for i in range(iterations):
    cmd = ['./' + "phods"]
    value = float(check_output(cmd).decode())
    phods.append(value)
    cmd = ['./' + "phods_v2"]
    value = float(check_output(cmd).decode())
    phods_v2.append(value)
    cmd = ['./' + "phods_v3"]
    value = float(check_output(cmd).decode())
    phods_v3.append(value)
    cmd = ['./' + "phods_v4"]
    value = float(check_output(cmd).decode())
    phods_v4.append(value)
    cmd = ['./' + "phods_v4_2d", str(144), str(8)]
    value = float(check_output(cmd).decode())
    phods_v4_2d.append(value)
    df = pd.DataFrame({'phods': phods, 'phods_v2': phods_v2, 'phods_v3':
    phods_v3, 'phods_v4': phods_v4, 'phods_v4_2d': phods_v4_2d})
    boxplot = df.boxplot(column=['phods', 'phods_v2', 'phods_v3', 'phods_v4',
    'phods_v4_2d'])
    plt.title("Box plot")
    plt.xlabel("Versions")
    plt.ylabel("time(s)")
    plt.show()
```

In the above diagram, 5 boxes are shown, each of which is related to a transformation we have applied in the previous questions. More specifically:

1. phods: original code (without transformations)
2. phods_v2: Loop fusion implementation (Merge)
3. phods_v3: Loop Unrolling implementation



4. phods_v4: Implementation of optimal block B=16

5. phods_v5: Implementation of optimal pair Bx=144, By=8

Comments:
In each new version (resulting from a transformation), we observe that we have an improvement in runtime in both the average price as well as in the min, max values. Comparing the boxes with the original, it can be seen that the range of runtime values has been reduced considerably.

2nd Issue: Automated Code Optimization

1. After successfully installing the Orio tool, the we used it to optimize the tables.c code, whose function involves simple accesses and operations with tables. Initially, we measured the execution time of . The time measurement was done with function gettimeofday with microsecond precision. We executed the code 10 times and measured the average, maximum and minimum time execution time. We again used the time.py script we had we created in question 2. The results of the measurement are shown below:

Minimum (s)	Average (s)	Maximum (s)
0.045432	0.048513	0.059486

2. We adapted the tables_orio.c file to implement the loop unrolling transformation to the code fragment of query 1. For each of the following modes listed on the Orio (Exhaustive, Randomsearch, Simplex) we performed Design Space Exploration to find the appropriate loop unrolling factor for this piece of code. Because for $N = 10^8$, Orio's compile displayed the specific error, we reduced N $N = 10^7$. The initial loop in table.c is as follows:

```
for (i=0; i<=N-1; i++
{
    //This loop needs to be modified after Orio's execution...
    y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i];
}
```

Exhaustive: we have two cases.

```
for (int loop_loop_38=0; loop_loop_38 < 1; loop_loop_38++) {
    int i;
    for (i = 0; i <= N - 2; i = i + 2) {
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
        y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 *
   x3[(i + 1)];
    }
    for (i = N - ((N - (0)) % 2); i <= N - 1; i = i + 1)
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
}
```

Listing 3: For $N \leq 1000$ we have Unroll Factor 2

```
for (int loop_loop_38=0; loop_loop_38 < 1; loop_loop_38++) {
    int i;
    for (i = 0; i <= N - 3; i = i + 3) {
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
        y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 * x3[(i + 1)];
        y[(i + 2)] = y[(i + 2)] + a1 * x1[(i + 2)] + a2 * x2[(i + 2)] + a3 * x3[(i + 2)];
    }
    for (i = N - ((N - (0)) % 3); i <= N - 1; i = i + 1)
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
}
```

Listing 4: Για $N \leq 10000000$ έχουμε Unroll Factor 3

Randomsearch: we used the following parameters:

```
def search {
    arg algorithm = 'Randomsearch';
    arg time_limit = 10;
    arg total_runs = 10;
}
```

```
for (int loop_loop_40=0; loop_loop_40 < 1; loop_loop_40++) {
    int i;
    for (i = 0; i <= N - 5; i = i + 5) {
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
        y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 * x3[(i + 1)];
        y[(i + 2)] = y[(i + 2)] + a1 * x1[(i + 2)] + a2 * x2[(i + 2)] + a3 * x3[(i + 2)];
        y[(i + 3)] = y[(i + 3)] + a1 * x1[(i + 3)] + a2 * x2[(i + 3)] + a3 * x3[(i + 3)];
        y[(i + 4)] = y[(i + 4)] + a1 * x1[(i + 4)] + a2 * x2[(i + 4)] + a3 * x3[(i + 4)];
    }
    for (i = N - ((N - (0)) % 5); i <= N - 1; i = i + 1)
    y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
}
```

Listing 5: Για $N \leq 1000$ έχουμε Unroll Factor 5

```
for (int loop_loop_40=0; loop_loop_40 < 1; loop_loop_40++) {
int i;
for (i = 0; i <= N - 15; i = i + 15) {
    y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
    y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 * x3[(i + 1)];
    y[(i + 2)] = y[(i + 2)] + a1 * x1[(i + 2)] + a2 * x2[(i + 2)] + a3 * x3[(i + 2)];
    y[(i + 3)] = y[(i + 3)] + a1 * x1[(i + 3)] + a2 * x2[(i + 3)] + a3 * x3[(i + 3)];
    y[(i + 4)] = y[(i + 4)] + a1 * x1[(i + 4)] + a2 * x2[(i + 4)] + a3 * x3[(i + 4)];
    y[(i + 5)] = y[(i + 5)] + a1 * x1[(i + 5)] + a2 * x2[(i + 5)] + a3 * x3[(i + 5)];
    y[(i + 6)] = y[(i + 6)] + a1 * x1[(i + 6)] + a2 * x2[(i + 6)] + a3 * x3[(i + 6)];
    y[(i + 7)] = y[(i + 7)] + a1 * x1[(i + 7)] + a2 * x2[(i + 7)] + a3 * x3[(i + 7)];
    y[(i + 8)] = y[(i + 8)] + a1 * x1[(i + 8)] + a2 * x2[(i + 8)] + a3 * x3[(i + 8)];
```

```
    y[(i + 9)] = y[(i + 9)] + a1 * x1[(i + 9)] + a2 * x2[(i + 9)] + a3 * x3[(
    i + 9)];
    y[(i + 10)] = y[(i + 10)] + a1 * x1[(i + 10)] + a2 * x2[(i + 10)] + a3 *
    x3[(i + 10)];
    y[(i + 11)] = y[(i + 11)] + a1 * x1[(i + 11)] + a2 * x2[(i + 11)] + a3 *
    x3[(i + 11)];
    y[(i + 12)] = y[(i + 12)] + a1 * x1[(i + 12)] + a2 * x2[(i + 12)] + a3 *
    x3[(i + 12)];
    y[(i + 13)] = y[(i + 13)] + a1 * x1[(i + 13)] + a2 * x2[(i + 13)] + a3 *
    x3[(i + 13)];
    y[(i + 14)] = y[(i + 14)] + a1 * x1[(i + 14)] + a2 * x2[(i + 14)] + a3 *
    x3[(i + 14)];
    }
    for (i = N - ((N - (0)) % 15); i <= N - 1; i = i + 1)
    y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
}
/*@ end @*/
}
```

Listing 6: Για $N \leq 10000000$ έχουμε Unroll Factor 15

Simplex: In the case of the simplex algorithm, we used the the following parameters:

```
def search {
    arg algorithm = 'Simplex';
    arg time_limit = 10;
    arg total_runs = 10;
    arg simplex_local_distance = 2;
    arg simplex_reflection_coef = 1.5;
    arg simplex_expansion_coef = 2.5;
    arg simplex_contraction_coef = 0.6;
    arg simplex_shrinkage_coef = 0.7;
}
```

```
for (int loop_loop_45=0; loop_loop_45 < 1; loop_loop_45++) {
    int i;
    for (i = 0; i <= N - 4; i = i + 4) {
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
        y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 *
    x3[(i + 1)];
        y[(i + 2)] = y[(i + 2)] + a1 * x1[(i + 2)] + a2 * x2[(i + 2)] + a3 *
    x3[(i + 2)];
        y[(i + 3)] = y[(i + 3)] + a1 * x1[(i + 3)] + a2 * x2[(i + 3)] + a3 *
    x3[(i + 3)];
    }
    for (i = N - ((N - (0)) % 4); i <= N - 1; i = i + 1)
    y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
}
```

Listing 7: Για $N \leq 1000$ έχουμε Unroll Factor 4

```
for (int loop_loop_45=0; loop_loop_45 < 1; loop_loop_45++) {
    int i;
    for (i = 0; i <= N - 11; i = i + 11) {
        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
        y[(i + 1)] = y[(i + 1)] + a1 * x1[(i + 1)] + a2 * x2[(i + 1)] + a3 *
    x3[(i + 1)];
        y[(i + 2)] = y[(i + 2)] + a1 * x1[(i + 2)] + a2 * x2[(i + 2)] + a3 *
    x3[(i + 2)];
        y[(i + 3)] = y[(i + 3)] + a1 * x1[(i + 3)] + a2 * x2[(i + 3)] + a3 *
    x3[(i + 3)];
        y[(i + 4)] = y[(i + 4)] + a1 * x1[(i + 4)] + a2 * x2[(i + 4)] + a3 *
    x3[(i + 4)];
```

```
9         y[(i + 5)] = y[(i + 5)] + a1 * x1[(i + 5)] + a2 * x2[(i + 5)] + a3 *
    x3[(i + 5)];
10        y[(i + 6)] = y[(i + 6)] + a1 * x1[(i + 6)] + a2 * x2[(i + 6)] + a3 *
    x3[(i + 6)];
11        y[(i + 7)] = y[(i + 7)] + a1 * x1[(i + 7)] + a2 * x2[(i + 7)] + a3 *
    x3[(i + 7)];
12        y[(i + 8)] = y[(i + 8)] + a1 * x1[(i + 8)] + a2 * x2[(i + 8)] + a3 *
    x3[(i + 8)];
13        y[(i + 9)] = y[(i + 9)] + a1 * x1[(i + 9)] + a2 * x2[(i + 9)] + a3 *
    x3[(i + 9)];
14        y[(i + 10)] = y[(i + 10)] + a1 * x1[(i + 10)] + a2 * x2[(i + 10)] +
    a3 * x3[(i + 10)];
15     }
16     for (i = N - ((N - (0)) % 11); i <= N - 1; i = i + 1)
17        y[i] = y[i] + a1 * x1[i] + a2 * x2[i] + a3 * x3[i];
18     }
19 /*@ end @*/
```

Listing 8: Για $N \leq 10000000$ έχουμε Unroll Factor 11

Σε κάθε μία περίπτωση καταγράψαμε το unroll factor που προέκυψε από την εξερεύνηση. Είδαμε ότι διαφέρουν μεταξύ τους, καθώς προσεγγίσαμε το πρόβλημα με διαφορετικό αλγόριθμο.

Algorithm	Unroll factor
Exhaustive	2 and 3
Randomsearch	5 and 15
Simplex	4 and 11

3. After the transformed code has been produced for each of the three ways of exploring the design space, we replaced the the corresponding piece of code in the tables.c file. Obviously because we we have N = 10 we choose the second instances of the loops and for 7 three algorithms. We measured their execution time. The measurement of time was measured with the gettimeofday function with microsecond precision. We executed the code 10 times and measured the average average, maximum and minimum execution time (time.py). We present the results of each measurement.

	Minimum (s)	Average (s)	Maximum (s)
Exhaustive	0.042233	0.043519	0.046476
Randomsearch	0.042779	0.043983	0.048376
Simplex	0.042181	0.043321	0.047977

Based on the results of the above measurements, it is obvious that all three methods reduced the execution times significantly compared to their untransformed version. On average, the fastest was Simplex with Exhaustive following.