lab2 report

Objectives

- Reviewing/Working with the source code optimization techniques (Loop Invariant Code Motion-LICM, Reducing unnecessary Statements and Memory Access, Loop Unrolling, Code Inclining, Minimizing the impact of Pointers, etc)
- Reviewing/Working with the System Functions for Time in the source code, measuring the execution times

2 Applying Optimization on Source Code In this Lab, you will use `A2-code.c`, which contains various parts, and will apply different Program Optimization techniques on the source code (different parts) to analyse the code execution times.

After applying each optimization technique, changing the source code and finding the execution time (before and after optimization), record all the execution times in a proper table for further comparisons.

2.1 Loop Invariant Code Motion

In this section, you will use `A2-code.c`, partial code-1, processing vectors.

```
for (j = 0; j < 100; j++)
    for (i = 0; i < 100; i++) {
        t1[i] = t1[i] + t2[j];
    }
```

1. Use the function `clock()` before and after the original code-1 to find the execution time of this process
2. Change the code-1 using “**Loop Invariant Code Motion** (LICM) technique” Note – At this step only change the invariant part of the loop
3. Use the function `clock()` again before and after the changed code-1 to find the execution time of its process

4. Repeat step 3 for 5 times, every time with a new compilation and execution process from the beginning, finding the average of various execution times for code-1 after change; enter all times in the result table [4 mark]

Note – Feel free to increase the Vector size in the code (from step1), if this produces more clear execution time.

This is how I optimize the code, I used `ARRAY_SIZE=1000000`. Storing `t2[j]` as `tmp` reduce repetitive access to the `t2` array when doing addition in the inner loop.

```
int tmp = 0;
    for (j = 0; j < ARRAY_SIZE; j++) {
        tmp = t2[j];
        for (i = 0; i < ARRAY_SIZE; i++) {
            t1[i] = t1[i] + tmp;
        }
    }
```

Using LICM improves code performance from 7781103.6 to 6801089.2

```
Starting of the program, start_t = 602
Not using optimization
End of the program, end_t = 8078398
Time taken by CPU: 8077796

Starting of the program, start_t = 618
Not using optimization
End of the program, end_t = 7726330
Time taken by CPU: 7725712

Starting of the program, start_t = 625
Not using optimization
End of the program, end_t = 7697426
Time taken by CPU: 7696801

Starting of the program, start_t = 581
Not using optimization
End of the program, end_t = 7718321
Time taken by CPU: 7717740

Starting of the program, start_t = 522
Not using optimization
End of the program, end_t = 7687389
Time taken by CPU: 7686867
```

```
Starting of the program, start_t = 711
Using optimization
End of the program, end_t = 7169418
Time taken by CPU: 7168707

Starting of the program, start_t = 638
Using optimization
End of the program, end_t = 6873224
Time taken by CPU: 6872586

Starting of the program, start_t = 719
Using optimization
End of the program, end_t = 6629514
Time taken by CPU: 6628795

Starting of the program, start_t = 629
Using optimization
End of the program, end_t = 6634285
Time taken by CPU: 6633656

Starting of the program, start_t = 541
Using optimization
End of the program, end_t = 6702242
Time taken by CPU: 6701701
```

2.2 Reducing unnecessary Statements and Memory Access

In this section, you will use `A2-code.c`, partial code-2, processing vectors.

```
for (i = 0; i < 100; i++) {
    t1[i] = t1[i] + 1;
}
```

1. Use the function `clock()` before and after the original code-2 to find the execution time of this process
2. Change the code-2 using "Reducing unnecessary Statements and Memory Access technique"

   Note - At this step only change the access to Vector elements; note that each access by `V[index]` syntax is equal to several instructions to execute

3. Use the function `clock()` again before and after the changed code-2 to find the execution time of its process
4. Repeat step 3 for 5 times, every time with a new compilation and execution, finding the average of execution time for code-2 after change; enter all times in the result table [4 mark]

Note – Feel free to increase the Vector size in the code (from step1), if this produces more clear execution time.

```
int *ptr = t1;
for (int i = 0; i < ARRAY_SIZE; i++) {
    *ptr++ = *ptr + 1;
}
```

Using pointer reference improve the performance from 64.4 to 47.

Using pointer reference avoid accessing memory (load or store) by adding the address of the start of the vector with the index each time.

```
Starting of the program, start_t = 271
End of the program, end_t = 334
Time taken by CPU: 63

Starting of the program, start_t = 278
End of the program, end_t = 333
Time taken by CPU: 55

Starting of the program, start_t = 339
End of the program, end_t = 395
Time taken by CPU: 56

Starting of the program, start_t = 361
End of the program, end_t = 429
Time taken by CPU: 68

Starting of the program, start_t = 270
End of the program, end_t = 350
Time taken by CPU: 80
```

```
Starting of the program, start_t = 393
End of the program, end_t = 439
Time taken by CPU: 46

Starting of the program, start_t = 267
```

```
End of the program, end_t = 319
Time taken by CPU: 52

Starting of the program, start_t = 282
End of the program, end_t = 326
Time taken by CPU: 44

Starting of the program, start_t = 266
End of the program, end_t = 305
Time taken by CPU: 39

Starting of the program, start_t = 291
End of the program, end_t = 345
Time taken by CPU: 54
```

2.3 Loop Unrolling

```
for (i = 0; i < 100; i++) {
    t1[i] = t1[i] + 1;
}
```

In this section, you will use `A2-code.c`, partial code-2, processing vectors.

1. Use the function `clock()` before and after the original code-2 to find the execution time of this process
2. Change the code-2 using "Loop Unrolling" Note - At this step only change the loop statements preferably into two versions: accessing to 3 and 4 elements of the Vector in each loop iteration
3. Use the function `clock()` again before and after the changed code-2 (for both versions) to find the execution time of its process
4. Repeat step 3 for 5 times (for both versions), every time with a new compilation and execution, finding the average of execution time for code-2 after change; enter all times in the result table [4 mark]

Note – Feel free to increase the Vector size in the code (from step1), if this produces more clear execution time.

```
#define UNROLL_FACTOR 4
for (i = 0; i < ARRAY_SIZE - UNROLL_FACTOR; i += UNROLL_FACTOR) {
```

```
    t1[i]     = t1[i] + 1;
    t1[i + 1] = t1[i + 1] + 1;
    t1[i + 2] = t1[i + 2] + 1;
    t1[i + 3] = t1[i + 3] + 1;
}
for (; i < ARRAY_SIZE; i++) {
    t1[i] = t1[i] + 1;
}
```

Using loop unrolling improve the performance from 64.4 to 40.

64.4 is using the same one from 2.2

```
Starting of the program, start_t = 398
End of the program, end_t = 444
Time taken by CPU: 46

Starting of the program, start_t = 225
End of the program, end_t = 265
Time taken by CPU: 40

Starting of the program, start_t = 205
End of the program, end_t = 243
Time taken by CPU: 38

Starting of the program, start_t = 215
End of the program, end_t = 253
Time taken by CPU: 38

Starting of the program, start_t = 189
End of the program, end_t = 227
Time taken by CPU: 38
```

2.4 Code Inlining

In this section, you will use `A2-code.c`, partial code-3, processing vectors.

```
for (i = 0; i < 10000; i++) {
    x     = abs(rand());
    t3[i] = func1(i, x);
    sum += t3[i];
}
```

1. Use the function `clock()` before and after the original code-3 to find the execution time of this process
2. Change the code-3 using "Code Inlining" Note - At this step only change the access to function `func1()` result; each access to function syntax is equal to several instructions to execute
3. Use the function `clock()` again before and after the changed code-3 to find the execution time of its process
4. Repeat step 3 for 5 times, every time with a new compilation and execution, finding the average of execution time for code-3 after change; enter all times in the result table [4 mark]

Note – Feel free to increase/decrease the Vector size in the code (from step1), if this produces more clear execution time.

```
for (i = 0; i < 10000; i++) {
    x     = abs(rand());
    t3[i] = (i + x) / 3;
    sum += t3[i];
}
```

Using code inlining improve the performance from 98.6 to 88.4.

```
Starting of the program, start_t = 386
End of the program, end_t = 498
Time taken by CPU: 112

Starting of the program, start_t = 328
End of the program, end_t = 431
Time taken by CPU: 103

Starting of the program, start_t = 296
End of the program, end_t = 393
Time taken by CPU: 97

Starting of the program, start_t = 239
End of the program, end_t = 329
Time taken by CPU: 90

Starting of the program, start_t = 234
End of the program, end_t = 325
Time taken by CPU: 91
```

```
Starting of the program, start_t = 357
End of the program, end_t = 453
Time taken by CPU: 96

Starting of the program, start_t = 222
End of the program, end_t = 309
Time taken by CPU: 87

Starting of the program, start_t = 215
End of the program, end_t = 303
Time taken by CPU: 88

Starting of the program, start_t = 178
End of the program, end_t = 262
Time taken by CPU: 84

Starting of the program, start_t = 174
End of the program, end_t = 261
Time taken by CPU: 87
```