# Lab 2 - Exercise 2

## 1 Experimental Setup

The following C program is used in this experiment:

```
for (int i = 0; i < N; i++) {
    x = a * b + x;
    y = a * b + y;
}
```

The loop performs two independent floating-point accumulation streams. The program is compiled and executed using:

- `-O0`: no compiler optimizations
- `-O2`: aggressive compiler optimizations

Execution time is measured using `clock()`.

## 2 Question 1: Execution Time Comparison

The program was compiled and executed as follows:

```
gcc -O0 exercice3.c -o ex_O0
gcc -O2 exercice3.c -o ex_O2
```

The measured execution times are shown in Table 1.

| Version | Optimization Level | Time (s) |
|---|---|---|
| Original code | -O0 | 0.297 |
| Original code | -O2 | 0.086 |

Table 1: Measured execution times

The speedup achieved by `-O2` over `-O0` is:

$$\text{Speedup} = \frac{0.297}{0.086} \approx 3.45$$

**Answer:** Compilation with `-O2` reduces execution time by approximately **3.45**$\times$ compared to `-O0`.

## 3 Question 2: Optimizations Performed at `-O2`

Analysis of the generated assembly code reveals several important optimizations at `-O2`:

### 3.1 Redundant Computation Elimination

The expression `a*b` is constant inside the loop. At `-O0`, it is recomputed for each update of `x` and `y`. At `-O2`, the compiler computes it once and reuses the result.

### 3.2 Register Allocation

At `-O0`, variables such as `x`, `y`, and the loop counter are frequently loaded from and stored to memory. At `-O2`, these variables are kept in registers, significantly reducing memory access latency.

### 3.3 Loop Transformations

The optimized assembly shows that the compiler performs loop unrolling. Each loop iteration processes multiple updates, reducing:

- the number of branch instructions,
- loop counter updates,
- condition checks.

### 3.4 Instruction Scheduling and ILP

Independent floating-point additions are reordered and scheduled to overlap execution. This allows the CPU pipeline to remain busy and hides instruction latency, increasing instruction-level parallelism.

**Answer:** Compared to `-O0`, `-O2` applies redundant computation elimination, register allocation, loop unrolling, and instruction scheduling to exploit ILP and reduce execution overhead.

## 4 Question 3: Manual Optimization vs Compiler Optimization

A manually optimized version of the code was implemented by computing `a*b` once before the loop:

```
const double t = a * b;
for (int i = 0; i < N; i++) {
    x += t;
    y += t;
}
```

This version was compiled with `-O0` and executed multiple times.

### 4.1 Measured Results

Average execution time for the manually optimized version:

$$T_{\text{manual},-O0} \approx 0.292 \text{ s}$$

| Version | Optimization Level | Time (s) |
|---|---|---|
| Original code | -O0 | 0.297 |
| Manual optimized code | -O0 | 0.292 |
| Original code | -O2 | 0.086 |

Table 2: Comparison of manual and automatic optimizations

### 4.2 Analysis

Manual optimization yields a small improvement of about 2% compared to the original `-O0` version. However, the `-O2` optimized code remains approximately **3.4× faster**.

This demonstrates that while manual optimizations can remove obvious inefficiencies, the compiler applies multiple advanced optimizations simultaneously that are difficult to reproduce manually at the source-code level.

**Answer:** Manual optimization at `-O0` provides only marginal improvement, whereas compiler optimization at `-O2` achieves substantially better performance.

## 5 Conclusion

This experiment demonstrates the effectiveness of compiler optimizations in exploiting instruction-level parallelism. While manual code restructuring can slightly improve performance, modern compilers at `-O2` outperform manual optimizations by combining register allocation, loop transformations, and instruction scheduling.