Runtime Comparison between native C- and Assembler-Code

Basic Setup

For this task we had to develop a function which takes a large array randomly filled with “char”s. The programming language we had to use was pure C. At first we had to implement a function in native C-Code and afterwards trying to optimize this function with the usage of Inline-Assembly-Code.

The method written in C looks like this:

```c
static void toupper_simple(char * text) {
	int i;
	for (i = 0; text[i] != '\0'; i++){

		char ch = text[i];
		if (!isupper(text[i]))
		{
			text[i] = ch - 0x20;
		}
	}
}
```

To measure our results we implemented a method which is able to count the time passed during runtime of another method.

```c
static inline
double gettime(void) {

	struct timeval mytimeval;
	time_t curtime;

	gettimeofday(&mytimeval, NULL);
	curtime = mytimeval.tv_sec;
	double exactpassedtime = (double)((mytimeval.tv_sec * 1000)+ mytimeval.tv_usec);
	return exactpassedtime;
}
```

Optimized Method

After this basic setup we are ready to optimize our “toupper”-method and afterwards compare the two function in regard to the runtime.

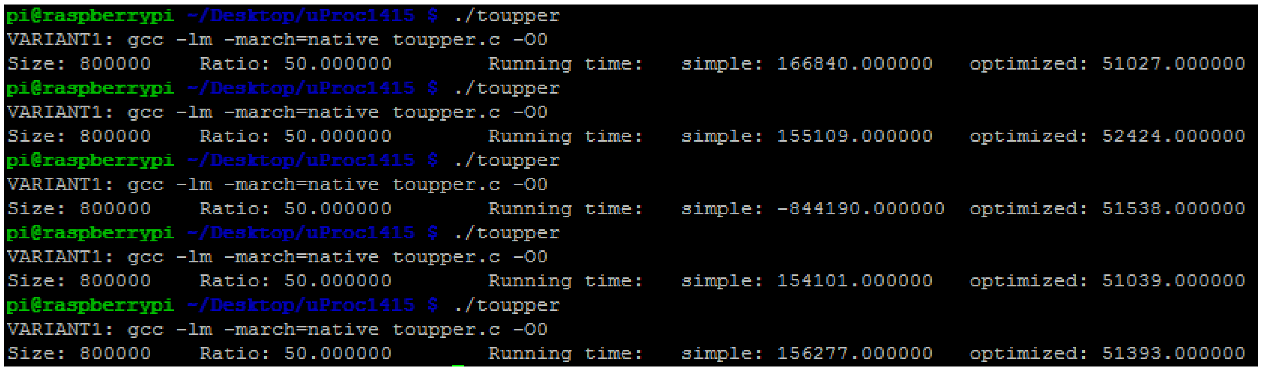
The optimized method looks as follows:

```
static void toupper_optimized(char * text) {
       int i;
       int result;

       for( i = 0; text[i] != '\0'; i++ )
       {
              __asm__("mov R4,%[R7] \n"
                      "CMP R4,#97 \n"
                      "SUBGE R4,#0x20"
                     : [R4] "=r" (result)        // Register R4 binded to result
                     : [R7] "r" (text[i]));      // Char at text[i] loaded to R7
       text[i] = (char) result;
       }

}
```

Results

As seen above we just optimized our inner for-loop with Assembly-Code. But even this tiny difference between the native C-Code and our optimized version changed the runtime a lot.

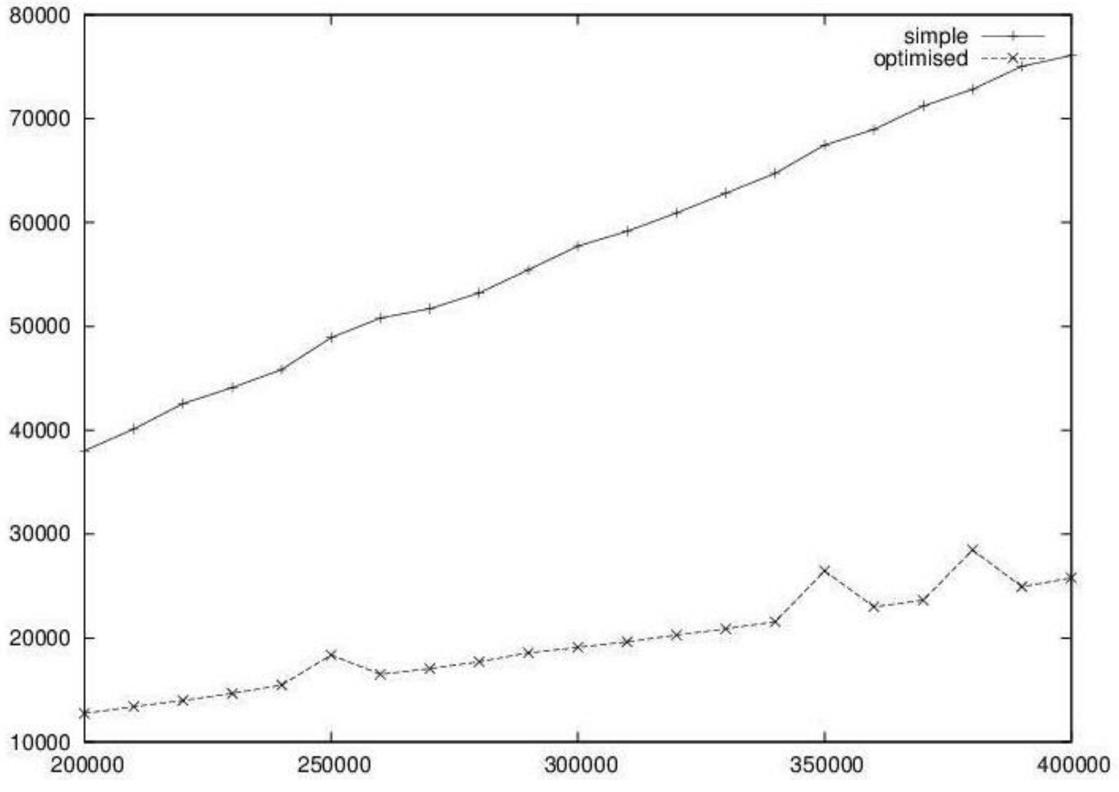
The results in a few test scenarios show that our optimized version is about three times faster than the original version.



```
pi@raspberrypi ~/Desktop/uProc1415 $ ./toupper
VARIANT1: gcc -lm -march=native toupper.c -O0
Size: 800000    Ratio: 50.000000        Running time:   simple: 166840.000000   optimized: 51027.000000
pi@raspberrypi ~/Desktop/uProc1415 $ ./toupper
VARIANT1: gcc -lm -march=native toupper.c -O0
Size: 800000    Ratio: 50.000000        Running time:   simple: 155109.000000   optimized: 52424.000000
pi@raspberrypi ~/Desktop/uProc1415 $ ./toupper
VARIANT1: gcc -lm -march=native toupper.c -O0
Size: 800000    Ratio: 50.000000        Running time:   simple: -844190.000000  optimized: 51538.000000
pi@raspberrypi ~/Desktop/uProc1415 $ ./toupper
VARIANT1: gcc -lm -march=native toupper.c -O0
Size: 800000    Ratio: 50.000000        Running time:   simple: 154101.000000   optimized: 51039.000000
pi@raspberrypi ~/Desktop/uProc1415 $ ./toupper
VARIANT1: gcc -lm -march=native toupper.c -O0
Size: 800000    Ratio: 50.000000        Running time:   simple: 156277.000000   optimized: 51393.000000
```

For getting more meaningful measuring results we plotted a graph where you can see the runtime in respect to the number of checked chars. The x-axis represent the number of checked chars whereas the y-axis displays the runtime of the method for a given number of chars. As you can see the runtime for the optimized code rises slower. At a number of 400,000 chars the runtime of the simple version is about 3 times slower than the optimized version.

To sum it up: Using Inline-Assembly-Code for code passages which are execute multiple times is very useful to increase the perform of one’s program.



80000
70000
60000
50000
40000
30000
20000
10000
200000
250000
300000
350000
400000
simple
optimised