Question 3

Using the C Programming language, write a program that sums an array of 50 elements. Next, optimize the code using loop unrolling. Loop unrolling is a program transformation that reduces the number of iterations for a loop by increasing the number of elements computed on each iteration. Generate a graph of performance improvement. Tip: Figure 5.17 in the textbook provides an example of a graph depicting performance improvements associated with loop unrolling. [30 marks]

Original program sumarray:

```c
int sumarray(int input[AMOUNT])
{
    int i;
    int acc = 0;
    for (i = 0; i < AMOUNT; i++)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc += input[i];
    }
    return acc;
}
```

sumarray1 which uses 2 x 1 unrolling:

```c
int sumarray1(int input[AMOUNT])
{
    int i;
    int acc = 0;
    int limit = AMOUNT-1;

    for (i = 0; i < limit; i+=2)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc = (acc + input[i]) + input[i+1];
    }

    for (; i < AMOUNT; i++)
        acc += input[i];

    return acc;
}
```

sumarray2 which uses 2 x 1 unrolling with re-association:

```
int sumarray2(int input[AMOUNT])
{
    int i;
    int acc = 0;
    int limit = AMOUNT-1;

    for (i = 0; i < limit; i+=2)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc += (input[i] + input[i+1]);
    }

    for (; i < AMOUNT; i++)
        acc += input[i];

    return acc;
}
```

sumarray3 which uses 5 x 1 unrolling and re-association:

```
int sumarray3(int input[AMOUNT])
{
    int i;
    int acc = 0;
    int limit = AMOUNT-4;

    for (i = 0; i < limit; i+=5)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc += (input[i] + input[i+1]);
        acc += (input[i+2] + input[i+3]);
        acc += input[i+4];
    }

    for (; i < AMOUNT; i++)
        acc += input[i];

    return acc;
}
```

sumarray4 which uses 5 x 5 unrolling:

```
int sumarray4(int input[AMOUNT])
{
    int i;
    int acc0 = 0;
    int acc1 = 0;
    int acc2 = 0;
    int acc3 = 0;
    int acc4 = 0;
    int limit = AMOUNT-4;

    for (i = 0; i < limit; i+=5)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc0 += input[i];
        acc1 += input[i+1];
        acc2 += input[i+2];
        acc3 += input[i+3];
        acc4 += input[i+4];
    }

    for (; i < AMOUNT; i++)
        acc0 += input[i];

    return acc0 + acc1 + acc2 + acc3 + acc4;
}
```

sumarray5 which uses 10 x 10 unrolling:

```
int sumarray5(int input[AMOUNT])
{
    int i;
    int acc0 = 0;
    int acc1 = 0;
    int acc2 = 0;
    int acc3 = 0;
    int acc4 = 0;
    int acc5 = 0;
    int acc6 = 0;
    int acc7 = 0;
    int acc8 = 0;
    int acc9 = 0;
    int limit = AMOUNT-4;

    for (i = 0; i < limit; i+=10)
    {
        int g;
        for (g = 0; g < DELAY; g++);
        acc0 += input[i];
        acc1 += input[i+1];
        acc2 += input[i+2];
        acc3 += input[i+3];
        acc4 += input[i+4];
        acc5 += input[i+5];
        acc6 += input[i+6];
        acc7 += input[i+7];
        acc8 += input[i+8];
        acc9 += input[i+9];
    }

    for (; i < AMOUNT; i++)
        acc0 += input[i];

    return (acc0 + acc1 + acc2 + acc3 + acc4 + acc5 + acc6 + acc7 + acc8 + acc9);
}
```

main:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 1000
#define AMOUNT 50
#define DELAY 0xfffffff
int main ()
{
    int array[AMOUNT];
    int i;
    srand(time(NULL));
    for (i = 0; i < AMOUNT; i++)
        array[i] = rand() % MAX; // use remainder operator to limit the size.

    int sum = sumarray(array);
    printf("The sum of the values of the array is %d\n", sum);

    int sum1 = sumarray1(array);
    printf("The sum of the values of the array is %d\n", sum1);

    int sum2 = sumarray2(array);
    printf("The sum of the values of the array is %d\n", sum2);

    int sum3 = sumarray3(array);
    printf("The sum of the values of the array is %d\n", sum3);

    int sum4 = sumarray4(array);
    printf("The sum of the values of the array is %d\n", sum4);

    int sum5 = sumarray5(array);
    printf("The sum of the values of the array is %d\n", sum5);

    exit (0);
}
```

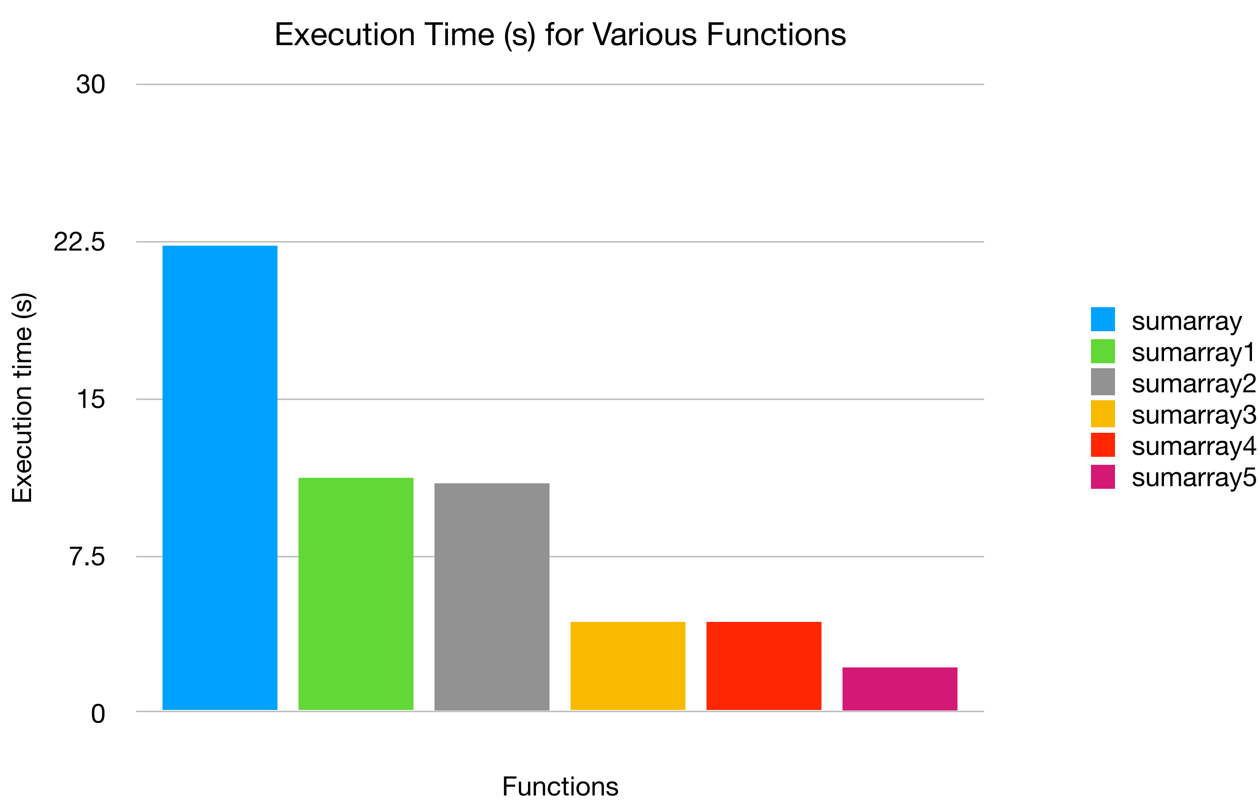
Execution results:

```
MichaelKuby@Michaels-iMac Question 3 % gcc-11 -O0 sumarray.c -o sumarray
MichaelKuby@Michaels-iMac Question 3 % ./sumarray
The sum of the values of the array is 24332
The sum of the values of the array is 24332
The sum of the values of the array is 24332
The sum of the values of the array is 24332
The sum of the values of the array is 24332
The sum of the values of the array is 24332
MichaelKuby@Michaels-iMac Question 3 % 
```

Analysis

By compiling with -pg and testing with GPROF we see some interesting results:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 40.66     22.31    22.31        1    22.31    22.31  sumarray
 20.38     33.49    11.18        1    11.18    11.18  sumarray1
 19.84     44.38    10.89        1    10.89    10.89  sumarray2
  7.85     48.68     4.31        1     4.31     4.31  sumarray4
  7.77     52.95     4.27        1     4.27     4.27  sumarray3
  3.86     55.07     2.12        1     2.12     2.12  sumarray5
```



The third column “self seconds” shows us the total run time in seconds for each function. Our original code sumarray takes over 22 seconds to execute. The fastest of our 5 functions is sumarray5, which utilizes 10 x 10 unrolling. The difference here, rounded to the hundredth, is 22.31 / 2.12 = 10.52, meaning we’ve achieved a speed up of over a factor of 10. Pretty impressive.

From sumarray to sumarray1, which utilizes 2 x 1 unrolling with suboptimal association, we see an immediately drastic improvement of about a factor of 2. Somewhat surprisingly we notice that the reassociation technique utilized by sumarray2 offers almost no improvement over sumarray1. Nevertheless, it does add some improvement, and should be utilized.

From sumarray2 to sumarray3 we see another drastic improvement. sumarray3 uses 5 x 1 unrolling and the proper associations. At this point we have hit the latency bound of the hardware.

Interestingly, sumarray4 has a very slight regression in performance in attempting to use multiple accumulators. It's hard for me to pinpoint exactly why this is happening. My first guess was that the regression in performance was because the numerous number of variables cannot all be held in registers, causing variables to be stored in memory. This seemed unlikely given the fact that there are only 5 accumulators and x86-64 hardware has 16 general purpose registers. The results of sumarray5 go to show that the issue was not spillage.

sumarray5 is our top performer, breaking through the latency limit and getting what I would assume is somewhere close to the throughput limit. sumarray5 uses 10 x 10 unrolling, so it's speed is achieved via the use of a large number of accumulators, which exploits the functional capabilities of the systems hardware.