TDT4200 Problem Set 3
Optimization

Graded, percentage score
The assignment is due by Thursday 2012-09-27, at 20:00.

Answers should be submitted on It's Learning. Your answer should consist of **two** files: `fft.c` for the programming task, and one report file (in .pdf) for the theory questions. The code should compile and run on `clustis3`. It is recommended that the code is readable with sensible comments. **Please** use the compute nodes when running large problem sizes (runs needing more than one second).

Task 1 - Theory (10 %)

a) Mention 3 different ways to reduce the number of branches.

b) How would you improve the performance of a long switch statement?

c) Mention 3 different ways to reduce the number of cache misses.

d) Mention 4 strategies of implementing SIMD in your code, and their advantages and disadvantages.

Task 2 - Code optimization (10 %)

Task 2.1

The following code snippet calculates $\sin(x)$ using the Taylor series expansion

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots .$$

It uses an additional function that calculates $n!$ (the factorial of n, defined as $n! = 1 \cdot 2 \cdot \ldots \cdot n$).

```
 /* calculate n! */
double factorial(int n) {
    if(n<2)
        return 1.0;
    else
        return (double)n*factorial(n-1);
}

/* calculate sin(x) using Taylor series expansion */
double slow_sin(double x) {
    double r=0; int i;
    for(i=0;i<100;i++) {
        if(i%2==0) {
            r += pow(x, i*2+1) / factorial(i*2+1);
        }
        else {
            r -= pow(x, i*2+1) / factorial(i*2+1);
        }
    }
    return r;
}
```

Mention 4 different ways to speed up the above code. Don't answer with code only, but give explanations.

Task 2.2

The following function takes in a 2D array of size 1024×1024 and calculates the sum of all elements. Explaing why the function is slower than it could have been.

```
double matrix_sum(double a[1024][1024]) {
    int i,j; double r=0;
    for(i=0;i<1024;i++){
        for(j=0;j<1024;j++) {
            r += a[j][i];
        }
    }
    return r;
}
```

Task 3 - Sequential optimization (80 %)

The purpose of this task is to program a function which calculates the Fourier transform of a one-dimensional array. This function should run as fast as possible.

The code provided in the archive `ps3.tar.gz` generates an array with complex numbers. This array is then transformed using a function which is chosen from the command line. Two functions are already implemented in the skeleton code: a naïve implementation of discrete Fourier transform (DFT) (with time complexity $\Theta(n^2)$) and a naïve implementation of fast Fourier transform (FFT) (with time complexity $\Theta(n \log n)$). The call to the chosen function will be timed, and the run time is reported in $\frac{\text{FLOP}}{1000 \text{ cycles}}$ (number of floating point operations per 1000 clock cycles). It also checks that the chosen function creates an answer which is approximately equal to that produced by the FFTW library.

The syntax of the program is:

```
./fft n a
```

where n is the size of the array to be transformed (n must be a power of two, e.g. 512, 1024 etc.), and a denotes which algorithm to use:

1 Naïve DFT

2 Naïve FFT

3 Your code

4 FFTW library

Your task is to write a function `my_fft()` which should be as fast as possible. Solutions should not rely on any external libraries or code not written by yourself. Your code should be sequential, and run on a single core (i.e no pthread, OpenMP, MPI etc.). You are allowed (and encouraged) to use intrinsics and inline assembly.

Your program should work for any reasonable input size n (between 1 and 1048576).

The grade breakdown is as follows:

- The program returns correct answer, and is faster than the provided naive fft implementation. (20%)

- The performanceof your program is at least X $\frac{\text{FLOP}}{1000\text{cycles}}$ (20%) (*Exact number will be announced shortly*)
- The last 40% of the grade is given based on the speed of your program compared to the fastest solutions submitted. The three fastest solutions (at least) will get the full 40%. (*More details will come shortly*)

The speed will be measured for two problem sizes: medium-sized and large. The input will be not generated with the same function as in the provided code.