IN3200/IN4200 Exercise Set 3

Exercise 1: How costly is a division operation?

Write a C program that uses the following function for carrying out the numerical integration of $\frac{4}{1+x^2}$ between $x_{\min}$ and $x_{\max}$ using a given number of "slices":

```
double numerical_integration (double x_min, double x_max, int slices)
{
  double delta_x = (x_max-x_min)/slices;
  double x, sum = 0.0;
  for (int i=0; i<slices; i++) {
    x = x_min + (i+0.5)*delta_x;
    sum = sum + 4.0/(1.0+x*x);
  }
  return sum*delta_x;
}
```

Please verify that with $x_{\min} = 0$ and $x_{\max} = 1$, the numerical integration of $\frac{4}{1+x^2}$ indeed approaches π when the number of "slices" is increased.

Suppose the floating-point division operation can not be pipelined, how will you use the C program to estimate the latency of a floating-point division in clock cycles?

Exercise 2: Dot-product and prefetching

The dot-product between two arrays can be computed by a `for`-loop:

```
double s = 0.;
for (i=0; i<N; i++)
  s = s + A[i]*B[i];
```

Suppose an imaginary **1GHz** CPU, which can do one load (or store), one multiplication and one addition per clock cycle. We assume that loop counting and branching come at no cost. The memory bus can transfer 3.2GBytes/sec. Assume that the latency to load one cache line from main memory is 100 clock cycles, and that four double precision numbers fit into one cache line.

(a) What is the expected performance for this loop kernel without data prefetching?

(b) Assuming that the CPU is capable of prefetching (loading cache lines from memory in advance so that they are present when needed), what is the required number of outstanding prefetches the CPU has to sustain in order to make this code bandwidth-limited, instead of latency-limited?

(c) How would this number change if the cache line were twice or four times as long?

(d) What is the expected performance of the dot-product calculation if we can assume that prefetching hides all the latency?

Exercise 3: Special implementation of `pow(x,100)`

The standard C math library function

```
double pow(double x, double y)
```

returns x raised to the power of y, that is x^y. This function is very general, but notoriously slow to execute on a computer.

Please write a special implementation of `pow(x,100)`, that is, when x is still a floating-point number but the power value y is fixed at integer 100. Only multiplications are needed, and please use as few multiplication operations as possible.

Write a C program to verify that your special implementation of `pow(x,100)` is indeed (much) faster than the standard `pow` function.

Exercise 4: Improving code performance

Explain why the following code will probably run very slow. How will you modify the code such that the same computation is done but the performance is considerably improved?

```
for (int i=0; i<n; i++) {
  c[i] = exp(1.0*i/n)+sin(3.1415926*i/n);
  for (int j=0; j<n; j++)
    a[j][i] = b[j][i] + d[j]*e[i];
}
```

We assume that both `a` and `b` are row-major 2D arrays of dimension $n \times n$, while `c`, `d`, `e` are 1D arrays of length n.