HOWTO WRITE FAST NUMERICAL CODE EXERCISE 2

Pascal Spörri
pascal@spoerri.io

March 13, 2013

1 Project Information

2 Microbenchmarks

For this exercise we had to benchmark several mathematical functions:

- $y = \sin(x)$, C Function: `y=sin(x)`
- $y = \log(x + 0.1)$, C Function: `y=log(x+0.1)`
- $y = e^x$, C Function: `y=exp(x)`
- $y = \frac{1}{x+1}$, C Function: `y=1.0/(x+1.0)`
- $y = x^2$, C Function: `y=x*x`

Since we benchmark on OSX we had the problem that we couldn't use `-march=corei7-avx` since the provided Apple assembler is unable to generate AVX code. Using a tip[1] we were able to replace the `as` program on our machines with the assembler from clang.

System Setup:

Compiler: gcc-4.7 (GCC) 4.7.2

Assembler: Apple clang version 4.1 (tags/Apple/clang-421.11.66) (based on LLVM 3.1svn)

Operating System: Mac OSX 10.8.2

CPU: Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz

We benchmarked our code with the following flags enabled: `-O3 -m64 -march=corei7-avx -fno-tree-vectorize`. We deliberately disabled vectorization since the automatic vectorization support for GCC is perceived as poor and we wanted to get explainable results.

[1] http://old.nabble.com/Re%3a-gcc,-as,-AVX,-binutils-and-MacOS-X-10.7-p32584737.html

Function	$x = 0$	$x = 0.9$	$x = 1.1$	$x = 4.12345$
$y = \sin(x)$	8.89	32.25	32.59	30.70
$y = \log(x + 0.1)$	22.26	20.81	20.95	25.93
$y = e^x$	11.13	20.68	23.19	23.48
$y = \frac{1}{x+1}$	6.26	10.36	10.46	10.63
$y = x^2$	1.61	1.50	1.62	1.64

Figure 1: Timings in cycles per mathematical function using `-O3 -m64 -march=corei7-avx -fno-tree-vectorize` with GCC 4.7.2.

2.1 Observations

- $y = \sin(x)$: We observe that we require significantly less cycles for $sin(0)$ than for the different function values of $\neq 0$. The library can make use of the approximation $\sin(\theta) \approx \theta$ for a significantly small theta.
- $y = \log(x + 0.1)$: We don't observe a significant change between the different function values.
- $y = e^x$: The CPU is able to make use of a direct computation for $x = 0$.
- $y = \frac{1}{x+1}$: The CPU is able to identify the special condition $\frac{1}{1}$.
- $y = x^2$: No change over the different inputs. The cycle count indicates pipelining.

3 Optimization Blockers

We were able to improve the performance of the code from 81 MFLOPs (size: 300) to 7.3 GFLOPs (size: 300).

System Setup:

Compiler: gcc-4.7 (GCC) 4.7.2

Assembler: Apple clang version 4.1 (tags/Apple/clang-421.11.66) (based on LLVM 3.1svn)

Compiler Options: `-m64 -march=corei7-avx -fno-tree-vectorize -O3`

Operating System: Mac OSX 10.8.2

CPU: Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz

```
void v10_inner_loop_unrolling(smat_t *a)
{
    int i, j;
    double x1,x2,y1,y2;
    double sum1, sum2;

    double *mat = a->mat;

```

```
    int n = a->n;
    int factor = 1;
    for(i = 0; i < n; i=i+2)
    {
        double cosp = cos(i)+1;
        double sinp = sin(factor*i);
        factor = -factor;

        int p = i+1;
        for(j = 0; j < n; j=j+2)
        {
            x1 = mat[i * n + j];
            y1 = mat[i * n + j+1];
            x2 = mat[p * n + j];
            y2 = mat[p * n + j+1];

            mat[i * n + j] = cosp * x1 + sinp * x2;
            mat[i * n + j+1] = cosp * y1 + sinp * y2;
            mat[p * n + j] = cosp * x2 - sinp * x1;
            mat[p * n + j+1] = cosp * y2 - sinp * y1;
        }
    }
}
```

Benchmarks:

Optimizations	**Optimized Compile Flags**		
	Size: 300	Size: 600	Average
Original Function	81 MFlops	75 MFlops	78 MFLOPs
Localized Loop Variables	67 MFlops	66 MFlops	67 MFLOPs
Inlineing f	76 MFlops	75 MFlops	75 MFLOPs
Replace Inner Functions	76 MFlops	73 MFlops	75 MFLOPs
Reorder Inner Loops	80 MFlops	82 MFlops	81 MFLOPs
Reorder cos and sin	283 MFlops	270 MFlops	277 MFLOPs
Move x1 and x2 into sum	399 MFlops	417 MFlops	407 MFLOPs
Restructure operations	419 MFlops	395 MFlops	407 MFLOPs
Inline get and set	2554 MFlops	2631 MFlops	2592 MFLOPs
Replace $x1 + cosp * x1$	2847 MFlops	4446 MFlops	3647 MFLOPs
Unroll inner loop	7344 MFlops	3912 MFlops	5628 MFLOPs

3.1 GCC