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

3.1 Code Variants

3.1.1 Original Code

The original code performed quite poorly. This is mainly due to multiple function calls and pointer references.

3.1.2 Localized Loop Variables

As a first step we have localized the loop variable n to avoid a memory lookup for each iteration.

3.1.3 Inlining f

Since the `f` function was only accessed once per loop iteration we inlined the function to get a better overview over the code.

3.1.4 Replacing the inner Functions

We removed the array from the inner loop.

3.1.5 Reordering the inner Loops

Looking at the code of the `get_elt` and `set_elt` function we noticed that the memory accesses were not coalesced. We therefore switched the loop ordering between the inner and outer loop.

3.1.6 Reordering cos and sin Functions

We move the computation of the `sin` and `cos` functions to the outer loop since they need to be computed only once per outer loop iteration.

3.1.7 Moving x1 and x2 into the sum

We replaced the memory accesses in the `sum1` and `sum2` with the already computed values `x1` and `x2`.

3.1.8 Restructuring operations

Using the fact that $\cos(x) = \cos(-x)$. We restructured the `sin` and `cos` computations such that the computations use less operations.

3.1.9 Inlining get and set

Since we can safely assume that our code is correct we inlined the get and set operations of the matrix. Therefore removing the bound checks from the code.

3.1.10 Replace `x1 + cosp * x1`

We replaced the computation of `x1 + cosp * x1` with `(cosp + 1)* x1` and added the `+1` to the outer loop.

3.1.11 Unroll inner loop

In order to get coalesced memory accesses we unrolled the inner loop. The code is provided in listing 1.

```
1  void v10_inner_loop_unrolling(smat_t *a)
2  {
3      int i, j;
4      double x1,x2,y1,y2;
5      double sum1, sum2;
6
7      double *mat = a->mat;
8
9      int n = a->n;
10     int factor = 1;
11     for(i = 0; i < n; i=i+2)
12     {
13         double cosp = cos(i)+1;
14         double sinp = sin(factor*i);
```

```
15          factor = -factor;
16
17          int p = i+1;
18          for(j = 0; j < n; j=j+2)
19          {
20              x1 = mat[i * n + j];
21              y1 = mat[i * n + j+1];
22              x2 = mat[p * n + j];
23              y2 = mat[p * n + j+1];
24
25              mat[i * n + j] = cosp * x1 + sinp * x2;
26              mat[i * n + j+1] = cosp * y1 + sinp * y2;
27              mat[p * n + j] = cosp * x2 - sinp * x1;
28              mat[p * n + j+1] = cosp * y2 - sinp * y1;
29          }
30      }
31  }
```

Listing 1: Improved code

3.2 Benchmarking

Due to benchmarking problems I decided to use a Linux operating system for benchmarking. We were able to improve the performance of the code from 74 MFLOPs (average) to 3.3 GFLOPs (average) getting a speedup of $44x$.

System Setup:

Compiler: gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3

Compiler Options: The two different compiler options that were used to test the code.

- Optimized Compile Flags: `-m64 -march=corei7-avx -fno-tree-vectorize -O3`
- Optimizations Disabled: `-m64 -O0`

Operating System: Ubuntu 12.04

CPU: Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz

We run the code with the two different compile flags on the setup above:

Optimizations	Optimized Compile Flags			Optimizations Disabled		
	Size: 300	Size: 600	**Average**	Size: 300	Size: 600	**Average**
Original Function	77 MFlops	72 MFlops	**74 MFLOPs**	29 MFlops	28 MFlops	**28 MFLOPs**
Localized Loop Variables	75 MFlops	69 MFlops	**72 MFLOPs**	30 MFlops	28 MFlops	**29 MFLOPs**
Inlineing f	76 MFlops	69 MFlops	**73 MFLOPs**	29 MFlops	28 MFlops	**29 MFLOPs**
Replace Inner Functions	76 MFlops	69 MFlops	**73 MFLOPs**	46 MFlops	43 MFlops	**45 MFLOPs**
Reorder Inner Loops	83 MFlops	82 MFlops	**82 MFLOPs**	49 MFlops	49 MFlops	**49 MFLOPs**
Reorder cos and sin	346 MFlops	340 MFlops	**343 MFLOPs**	120 MFlops	122 MFlops	**121 MFLOPs**
Move x1 and x2 into sum	500 MFlops	500 MFlops	**500 MFLOPs**	153 MFlops	157 MFlops	**155 MFLOPs**
Restructure operations	474 MFlops	500 MFlops	**487 MFLOPs**	153 MFlops	155 MFlops	**154 MFLOPs**
Inline get and set	2250 MFlops	2571 MFlops	**2411 MFLOPs**	500 MFlops	500 MFlops	**500 MFLOPs**
Replace $x1 + cosp * x1$	2250 MFlops	3600 MFlops	**2925 MFLOPs**	692 MFlops	667 MFlops	**679 MFLOPs**
Unroll inner loop	3000 MFlops	3600 MFlops	**3300 MFLOPs**	750 MFlops	750 MFlops	**750 MFLOPs**

Figure 2: Timings chart for the optimization blockers exercise.

We observed that the function "Unroll inner loop" (listing: 1) showed the best performance for both the unoptimized and the optimized compiler flags. We also noticed increased performance between the two different input sizes: $n = 300$ and $n = 600$. We believe this results from a better caching behaviour from my code with large matrices.

We also observe that the first 4 optimisations didn't increase the performance of the code. In combination with the other optimisations especially the Inlining of the `get set` operations gave the highest performance increase.