263-2300-00: How To Write Fast Numerical Code
Assignment 2: 80 points
Due Date: Th, March 16th, 17:00
http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring17/course.html
Questions: fastcode@lists.inf.ethz.ch

Submission instructions (read carefully):

- (Submission)
  Homework is submitted through the Moodle system https://moodle-app2.let.ethz.ch/course/view.php?id=3122. Before submission, you must enroll in the Moodle course.
- (Late policy)
  You have 3 late days, but can use at most 2 on one homework, meaning submit latest 48 hours after the due time. For example, submitting 1 hour late costs 1 late day. Note that each homework will be available for submission on the Moodle system 2 days after the deadline. However, if the accumulated time of the previous homework submissions exceeds 3 days, the homework will not count.
- (Formats)
  If you use programs (such as MS-Word or Latex) to create your assignment, convert it to PDF and name it homework.pdf. When submitting more than one file, make sure you create a zip archive that contains all related files, and does not exceed 10 MB. Handwritten parts can be scanned and included or brought (in time) to Alen's, Gagandeep's or Georg's office. Late homeworks have to be submitted electronically by email to the fastcode mailing list.
- (Plots)
  For plots/benchmarks, **provide (concise) necessary information for the experimental setup (e.g., compiler and flags) and always briefly discuss the plot and draw conclusions**. Follow (at least to a reasonable extent) the small guide to making plots from the lecture.
- (Code)
  When compiling the final code, ensure that you use optimization flags. **Disable auto-vectorization for this exercise when compiling**. Under Visual Studio you will find it under Config / Code Generator / Enable Enhanced Instructions (should be off). With `gcc` their are several flags: use `-mno-abm` (check the flag), `-fno-tree-vectorize` should also do the job and for `icc`, the flag `-no-vec`.
- (Neatness)
  5% of the points in a homework are given for neatness.

Exercises:

1. *Short project info (10 pts)* Go to the list of mile stones for the projects. If you have not done that yet, please register your project there. Read through the different points and fill in the first two with the following about your project (be brief):

   Point 1) An exact (as much as possible) but also short, problem specification.

   For example for MMM, it could be like this:

   Our goal is to implement matrix-matrix multiplication specified as follows:

   Input: Two real matrices A, B of compatible size, $A \in \mathbb{R}^{n \times k}$ and $B \in \mathbb{R}^{k \times m}$. We may impose divisibility conditions on n, k, m depending on the actual implementation.
   Output: The matrix product $C = AB \in \mathbb{R}^{n \times m}$.

   Give the name of the algorithm you plan to consider for the problem and a precise reference (e.g., a link to a publication plus the page number) that explains it.

   Point 2) A very short explanation of what kind of code already exists and in which language it is written.

2. *Optimization Blockers (20 pts)* Code needed
In this exercise, we consider the following short computation:

```
1 void slowperformance1(double *x, double *y, double *z, int n) {
2  double sum = 0;
3   for (int i = 0; i < n; i = i + 1)
4    for (int k = 0; k < 4; k++)
5     sum += x[i] * y[(i + k) % 5];
6   z[0] = sum; //we to this to avoid deadcode elimination
7 }
```

This is part of the suplied code.

(a) read and understand the supplied code. It enables you to register functions with the same signature, which will be timed in a microbenchmark fashion.

(b) Create new functions where you perform some loop unrolling and scalar replacement as discussed in the lecture to increase the performance. Explore at least three possible choices in this space, as different as possible.

(c) For every optimization you perform, create a new function in `comp.c` that has the same signature as *slowperformance1* and register it to the timing framework through the *register_function* function in `comp.cpp`. Let it run and, if it verifies, determine the performance.

(d) When done, rerun all code versions also with optimization flags turned off ($-O0$ in the Makefile).

(e) Create a table with the performance numbers. Two rows (optimization flags, no optimization flags) and as many columns as versions of *slowperformance1*. Briefly discuss the table.

(f) Submit your `comp.cpp` to Moodle.

What speedup do you achieve?

Solution:

The main purpose of this exercise is to observe the impact imposed by using different optimization flags. Additionally this exercise should also demonstrate that certain optimizations can not be performed by the compiler and therefore require manual intervention.

Without changing the complexity (which was not a strong requirement for this exercise), code that exhibits maximal performance could take the following form:

	-O3 -march		-O0		
	f/c	c/n	f/c	c/n	flops
`original`	0.500	15.998	0.171	46.686	8n
`inner loop unroll`	0.615	13.000	0.181	44.241	8n
`common factor`	0.385	12.999	0.167	29.944	5n
`pr comp`	0.445	4.499	0.177	11.316	2n
`pr comp outer loop unroll`	2.228[1]	0.898[1]	0.216	9.257	2n
`MaxPerformance`	1.538	0.650	0.171	5.833	5/4n

Table 1: Results submitted by Luca Corinzia, executed on a Skylake CPU: Intel Core i7 6700HQ 2.6 GHz, compiled using GNU `gcc`, running Ubuntu 16.04.2 LTS, vectorization disabled with `-fno-tree-vectorize`, cycles count scaled by the size of the input arrays. Note that the result obtained here[1] is quite higher than expected, and we believe that it could be as a result of having Turbo Boost enabled.

```c
void maxperformance_samecomplex(double *x, double *y, double *z, int n)
{
  unsigned short int i;
  double calc = 0;
  double sum0, sum1, sum2, sum3, sum4, sum5, sum6, sum7, sum8;
  double y0 = y[0], y1 = y[1], y2 = y[2], y3 = y[3], y4 = y[4];

  for (i = 0; i < n; i = i + 5)
  {
    sum0 = x[i] * y0;
    sum1 = x[i + 1] * y1;
    sum2 = x[i + 2] * y2;
    sum3 = x[i + 3] * y3;
    sum4 = x[i + 4] * y4;
    sum5 = sum0 + sum1 + sum2 + sum3 + sum4;

    sum0 = x[i] * y1;
    sum1 = x[i + 1] * y2;
    sum2 = x[i + 2] * y3;
    sum3 = x[i + 3] * y4;
    sum4 = x[i + 4] * y0;
    sum6 = sum0 + sum1 + sum2 + sum3 + sum4;

    sum0 = x[i] * y2;
    sum1 = x[i + 1] * y3;
    sum2 = x[i + 2] * y4;
    sum3 = x[i + 3] * y0;
    sum4 = x[i + 4] * y1;
    sum7 = sum0 + sum1 + sum2 + sum3 + sum4;

    sum0 = x[i] * y3;
    sum1 = x[i + 1] * y4;
    sum2 = x[i + 2] * y0;
    sum3 = x[i + 3] * y1;
    sum4 = x[i + 4] * y2;
    sum8 = sum0 + sum1 + sum2 + sum3 + sum4;

    calc = calc + sum5 + sum6 + sum7 + sum8;
  }
  // 20 mults, 20 adds - 40 flops / 5 iterations
  z[0] = calc;
}
```

As shown above, loops are unrolled and tiled such that it becomes possible to remove the modulo operation and to use several accumulators.

Many of you realized that one can of course also change the algorithmic complexity of the task. Some of these optimizations decrease performance, as they reduce the number of flops. As long as runtime gets decreased, such transformations are worth pursuing. A particularly nice solution was submitted by **Luca Corinzia** shown in Table 1.

The table clearly shows the relationship of performance and runtime. The runtime is given in cycles per length of the provided vector `x`. The accompanying code can be found here: Luca Corinzia's submission. It is possible to transform the computation to factor out some of the products in the style of $x[i] * y + x[i+1] * y = y * (x[i] + x[i+1])$. Luca, in addition, realized that part of computation can be precomputed and reused within the loop.

3. *Microbenchmarks(45 pts)* Code needed

   Write a program (without vector instruction, i.e, standard C, and compiled without autovectorization) that benchmarks the latency and throughput of a floating point multiplication and division instructions on doubles. Use the skeleton available provided and:

- Implement the functions provided in the skeleton:

```
void           microbenchmark_mode (microbenchmark_mode_t mode);
double         microbenchmark_get_mul_latency    ();
throughput_t   microbenchmark_get_mul_throughput ();
double         microbenchmark_get_div_latency    ();
throughput_t   microbenchmark_get_div_throughput ();
```

- Use `microbenchmark_mode` function bellow for initialization and test modes.
- Report your CPU, operating system and compiler.
- Report any optimization flags that you used.
- Report the results in cycles: latency and throughput, and calculate the gap.
- Generate `microbenchmark.s` - the assembly version of the benchmark.
- Submit only `microbenchmark.c` and `microbenchmark.s`.

Discussion:

(a) Can you reach the theoretical latency / throughput of the instructions?

(b) Does the multiplication instruction exhibit consistent latency / throughput on any input? Do you observe any changes if one operand is set to `0x0000000000000001`? Demonstrate this behaviour in the `MUL2_LATENCY` / `MUL2_THROUGHPUT` mode.

(c) Does the division instruction exhibit consistent latency / throughput on any input? Do you observe any changes if the divisor is set to 2? Demonstrate this behaviour in the `DIV2_LATENCY` / `DIV2_THROUGHPUT` mode.

(d) Do you observe any changes when flags like `-ffast-math -funsafe-math-optimizations` are enabled. If so, why?

Note that you might have to run the same instruction many times in order to get precise benchmark results. The code skeleton uses CMake to compile and uses Intel Performance Counter Monitor to measure performance on Intel based CPUs. If an Intel CPU is not available, it is possible to compile the code in "failback" mode that uses `RDTSC` only. In a nutshell, the compilation goes as follows:

```
cd microbenchmark
mkdir build
cd build
cmake ..
cd ..
cmake --build build --config Release
```

Then run it, using:

```
./bin/IntelPCM
```

For the failback mode:

```
cd microbenchmark
mkdir build
cd build
cmake -DRDTSC_FAILBACK=1 ..
cd ..
cmake --build build --config Release
```

The `zip` bundle contains detailed informations on the compilation steps in Linux / Windows / Mac OS X, available in `README.md`, as well as in the `doc` folder.

Solution:

A possible solution is available here.

```
================================================
GCC: -O3 -fno-vectorize
================================================

Measured mul latency     : 5.000000 cycles
Measured mul throughput  : 1.998135 f/c
Measured mul gap         : 0.500467 cycles

Measured div latency     : 20.016733 cycles
Measured div throughput  : 0.073630 f/c
Measured div gap         : 13.581333 cycles

Measured mul2 latency    : 130.000000 cycles
Measured mul2 throughput : 0.007671 f/c
Measured mul2 gap        : 130.367600 cycles

Measured div2 latency    : 10.138600 cycles
Measured div2 throughput : 0.124574 f/c
Measured div2 gap        : 8.027333 cycles
```

(a) `fast-math` disabled

```
================================================
GCC: -O3 -fno-vectorize -ffast-math
     -funsafe-math-optimizations
================================================

Measured mul latency     : 0.000000 cycles
Measured mul throughput  : 3.990423 f/c
Measured mul gap         : 0.250600 cycles

Measured div latency     : 0.668333 cycles
Measured div throughput  : 0.073547 f/c
Measured div gap         : 13.596667 cycles

Measured mul2 latency    : 0.000000 cycles
Measured mul2 throughput : 0.015328 f/c
Measured mul2 gap        : 65.239167 cycles

Measured div2 latency    : 0.342133 cycles
Measured div2 throughput : 0.124481 f/c
Measured div2 gap        : 8.033333 cycles
```

(b) `fast-math` enabled

Figure 1: Microbenchmarks executed on Haswell CPU - Intel(R) Core(TM) i7-4980HQ CPU @ 2.80GHz, running on Mac OS X El Capitan 10.11.6 (15G1217), compiled with `gcc` version: `Apple LLVM version 8.0.0 (clang-800.0.42.1)`

(a) By carefully crafting the `C` code, we are able to achieve latency and throughput which is close to the theoretical values of the instructions. To achieve as precise results as possible, we wrap the functions inside a loop, having many iterations, and average the obtained measurements.

While compiling with `-O3`, the compiler can perform various of optimizations. Most notable for this particular exercises are loop-invariant code motion and precomputation. To avoid precomputation in the compiler, we define global variables `x1` - `x16`, and we perform the initialization inside the `microbenchmark_mode` function. To avoid code motion we arrange the instructions inside the loop, creating artificial dependencies, to force the compiler to keep them inside the loop, or inside the measurement function.

When translated to machine code, the multiplication results with a `mulsd` (SSE) or `vmulsd` (AVX) instruction. According to the Intel 64 and IA-32 Architectures Optimization Reference Manual, the theoretical values of `mulsd` / `vmulsd` instruction is 5 cycles for latency and 0.5 cycles for the gap (or 2 multiplications per cycle). We observe consistent behaviour as indicated on Figure 1a.

When division is translated into machine code, it results with a `divsd` instruction. Theoretical values for `divsd` are 14 - 20 cycles for latency and 13 cycles for gap (or 0.076 divisions per cycle). We also observe consistent behavior for the `divsd` instruction, as indicated on Figure 1a. Note that to avoid overflows / underflows when testing the latency of `divsd` we choose values such that we perform division with `value` and then with `1 / value`. For throughput mode we set random numbers, and choose less iterations to avoid underflows.

(b) Note that `0x0000000000000001` does not correspond to 1.0, in fact it corresponds to a very little double precision floating point value that is close to `4.9406564584e-324`. This number is part of a special category of floating point numbers called "denormals". Denormals fill the underflow gap around zero in floating-point arithmetic.

Denormals have a huge penalties while being processed by the SSE / AVX FPUs. We observe this behaviour in the latency of the `mulsd` instruction in `MUL2_LATENCY` mode, having measured latency and throughtput of 130 cycles, shown on Figure 1a. Intel does not document this behaviour in the value of latency and throughput, since (after all) denormals represent a small subset of floating

```
==========================================
GCC: Inline assembly
==========================================

Measured mul latency     : 5.000450 cycles
Measured mul throughput  : 1.996577 f/c
Measured mul gap         : 0.500857 cycles

Measured div latency     : 20.017750 cycles
Measured div throughput  : 0.071277 f/c
Measured div gap         : 14.029714 cycles

Measured mul2 latency    : 141.481600 cycles
Measured mul2 throughput : 0.007662 f/c
Measured mul2 gap        : 130.509086 cycles

Measured div2 latency    : 10.112750 cycles
Measured div2 throughput : 0.124667 f/c
Measured div2 gap        : 8.021371 cycles
```

Figure 2: Microbenchmark using inline assembly, performed on Haswell CPU, Intel(R) Core(TM) i7-4980HQ CPU @ 2.80GHz, running on Mac OS X El Capitan 10.11.6 (15G1217), compiled with `gcc` version: `Apple LLVM version 8.0.0 (clang-800.0.42.1)`

point numbers. However this makes `mulsd` / `vmulsd` to have variable latency and throughput.

Depending on your compiler, you might not observe this behaviour. Namely Intel CPUs have special flags for SSE/AVX units, called denormals-are-zero (DAZ) and flush-to-zero (FTZ) flags. Some compilers enable this flags by default in the generated code, even in `-O0` mode. To learn more about denormals on SSE, check the tutorial, provided by Intel.

The denormal `0x0000000000000001` has a very interesting property. When multiplied with a very low number (for example 1.2), produces the same result. We use this property when crafting our C code measurements for both latency and throughput, forcing the compiler to execute multiplication using the same denormal operand on each iteration when measuring.

(c) Division - `divsd` has a variable latency and throughput, as well as many other CPU instructions. This is the reason why Intel provides a range for the latency value. The documentation also states 14 - 20 cycles for latency and 13 cycles for gap, but in reality both latency and gap have ranges. Real latency range for `divsd` is 10 - 20 cycles, and range for throughput is 8 - 14. Agner Fox has already documented this behavior available here.

The reason for the variable length is because `div` instruction is input dependant. Therefore in our experiment we reach the lower bounds for both latency and throughput, using 2 as a divisor. Note that 0.5 has also the same properties when used as a divisor, which we take advantage of when measuring.

(d) Once enabled, `-ffast-math -funsafe-math-optimizations` perform aggressive optimizations, that do not preserve IEEE 754 compliance. Some of the optimizations include reordering of instructions, loop unrolling, aggressive permutations, reciprocal approximations for division, common subexpression eliminations and assuming that all math is finite, ignoring denormals and `NaN`s. Note that the choice and order of these optimizations strictly depend on the initial C code, the compiler, and its version. However, if any of those is applied, will definitely break the normal and expected flow of our measuring infrastructure, leading to better performance. This is the reason why enabling those flags affects all measurements above, as shown in Figure 1b.

For the enthusiasts:

Assembly is not a requirment for this course. However, for the enthusiasts, we provide an assembly

version of the solution available here. The code uses inline assembly with AT&T syntax, which was tested on both `clang` (Apple) and `gcc` on Linux. Most of the explanation above remain the same, few things change:

- Obviously, since we inline the assembly code, `-ffast-math` and `-funsafe-math-optimizations` will not affect the results.
- Once performing the division measurements, we are able to get precise results for both latency and throughput, reaching the upper and lower bounds in both `DIV` and `DIV2` modes, shown in Figure 2. This is mainly because the assembly can be written without using the workarounds required to avoid loop invariant code motion, and precomutation.
- The penalty of processing denormals, can happen in both modes of the `mulsd` / `vmulsd` instruction, either in throughput mode, or latency mode. Without being bound by workarounds of loop invariant code motion, we demonstrate how this penalty is in fact even higher than 130 cycles in latency mode, as shown in Figure 2.