263-2300-00: How To Write Fast Numerical Code
Assignment 3: 100 points
Due Date: Th, March 30th, 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.
- (Neatness)
  5% of the points in a homework are given for neatness.
- (Using Student Labs for measurements - CAB H 56 / CAB H 57)
  The student labs are now setup with a special kernel image to ensure precise measurements. We have disabled Intel Turbo Boost Technology, disabled Intel Hyper-Threading Technology, isolated the last core from the Linux scheduler, and installed `Linux Perf` to enable access to core specific counters. This allows you to run your experiments such that you pin the executable to the highest available core. Note that this setup does not prevent preemption, but reduces the noise imposed by context switching. To access this setup, turn on (restart) the student machines and choose:

  ```
  [for 263-2300] Red Hat Enterprise Linux Workstation ...
  ```

Exercises:

1. *Matrix Computation (30 pts)* Code needed
   In this exercise, we consider the following matrix computation:

   ```
   void matrix_computation(double *C, double *A, double *B, int n) {
     int i,j,k;
     for (i = 0; i < n; i++){
       for (j = 0; j < n; j++){
         double sum = 0;
         for(k = 0; k < n; k++)
         sum = sum + fmin(A[n*i+k]*B[n*k+j],B[n*i+k]*A[n*k+j]);
         C[i][j] = sum;
       }
     }
   }
   ```

   where A, B and C are floating point matrices of size $N \times N$ where $N \in \{100, 200, \ldots, 1500\}$. All entries in A and B are randomly initialized. We provide a code skeleton containing a scalar implementation of the above computation in `src/comp_sisd.c`. Provide SSE and AVX based implementation of the computation in files `src/comp_sse.c` and `src/comp_avx.c` respectively. Try to optimize your code as much as possible. Make sure to validate your code using the validation framework provided in the skeleton.

What speedup do you achieve? Explain your observed performance.

How to compile

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

When running on CAB H 56 / CAB H 57 machines (or if you prefer Linux Perf on your machine):

```
mkdir build
cd build
cmake3 -DLINUX_PERF=1 ..
cd ..
cmake3 --build build --config Release
```

Note that the student labs required `cmake3` (since this project requires version 3.0.2+) while `cmake` is mapped to older version (2.8.12.2). If all else fails, the "failback" mode is still present:

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

Solution:

One possible solution is available here.

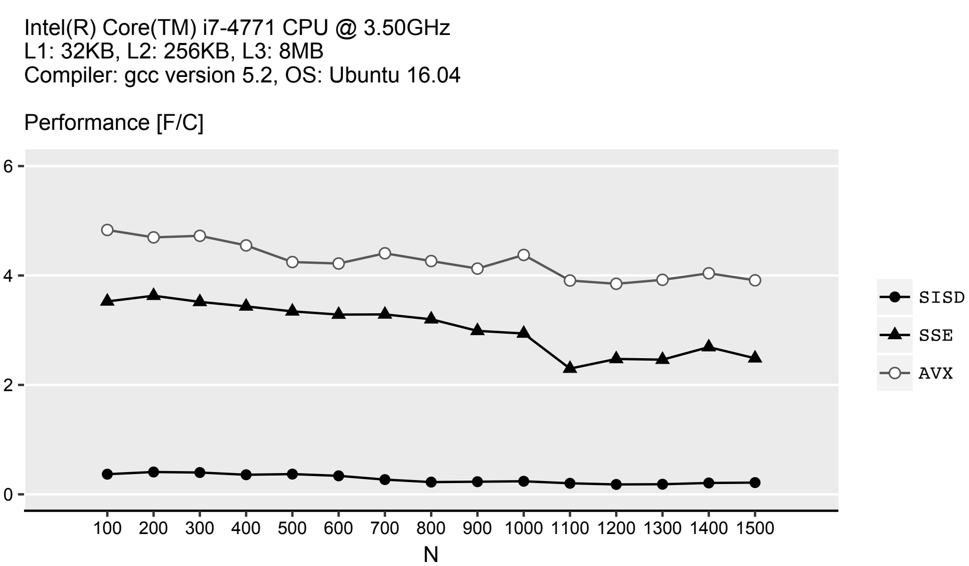

Figure 1: Performance of AVX, SSE vectorized and scalar version of `matrix_computation` function on a Haswell CPU (L1: 32 KB, L2: 256 KB and L3: 8192 KB). The code was compiled with `gcc` 5.2.1 with `O3` enabled, no FMAs and no auto-vectorization.

We perform blocking to speedup vectorized code. The code accesses two blocks from matrix `A`, `B` and one block from `C` per iteration. We use different block lengths BLA_{ik}, BLA_{kj}, BLB_{ik}, BLB_{kj} and $BLCij$ for matrices. For SSE version, $BLA_{ik} = BLB_{ik} = 400$, $BLA_{kj} = BLB_{kj} = 1000$, and

$BLC_{ij} = 1000$ whereas for AVX version, $BLA_{ik} = BLB_{ik} = 2000$, $BLA_{kj} = BLB_{kj} = 10000$ and $BLC_{ij} = 2000$. As a result, the blocks fit into L2 cache. We further unroll the blocked k-loop by a factor of 4 which boosts the performance. The speedup due to SSE and AVX varies between 10-15x and 13-20x respectively. The performance drops when one of the matrix does not fit in L3 cache.

2. *MVM (30 pts)* Code needed
   Your task is to vectorize the code for a 10×10 MVM $y = Ax$: Implement the function `vec_mvm10` in `mvm10.c` using **AVX** intrinsics; you can use any load or store instruction. Implement the function:

```
vec_mvm10(float const * A, float const * x, float * y);
```

   You can assume all arrays are 32-byte aligned. Submit your mvm10.c file including the vectorized variant. Make sure its vectorized using AVX!

   Solution

   One possible solution is available here.

3. *Power Function (35 pts)* Code needed
   Your task is to use **AVX** / **AVX2** and implement a power function **without a single** `if` / `switch` branch instruction and no recursion. Assume that the function is given in the form:

```
double power_avx (double x, uint32_t exponent);
```

   We provide a reference implementation of a `power_scalar (double x, uint32_t exponent)` function (recursive). Assume that $0 \leq$ `exponenent` $\leq 2^{32} - 1$. Follow the same compilation steps as in Exercise 1. Also assume that both performance and complexity optimizations are valid and do not use intrinsics such as `_mm256_exp_pd`, or any available intrinsics that perform exponential / logarithmic operations. Note that we do not expect binary compatibility upon validation, and relative error will be sufficient for this simple exercise.

   What speed up do you obtain? Can you beat the implementation in `math.h`? Why?

   Hint: You should be able to implement the function using addition, subtraction and multiplication from the arithmetic intrinsics, as well as any relational, logical and bitwise intrinsics.

   Solution:

   One possible solution is available here.

   We benefit from the fact that the exponent is a 32-bit integer number. Therefore, to calculate x^n, we calculate $x^{2^0}, x^{2^1}, x^{2^2}, x^{2^3}, \ldots x^{2^{31}}$ and use the bit-mask of the exponent to multiply those values. Therefore, our implementation runs in constant time, without any need for branching.

   Our implementation of `pow`, on average, is faster than the one of `libc` in the magnitude of about 30 cycles, with the provided input data, as shown in Figure 2.

   The `math.h` implementation of `pow` is input dependent, and depending on the compilation flags of the pre-compiled `libc`, it could result with various of implementation. In the very worst case scenario, a non-vectorized implementation of `pow` takes about 200 lines of code shown here. This means that on some input data, this function will be much faster, and much slower on other input data. Therefore the speed of `pow` on this particular input data will depend on the random number generator, `libc` version, as well as compiler version and OS calling conventions.

   The comparison is not fair, since the compiler can inline our implementation of `pow`, while `math.h` implementation will never be inlined. This means that the loss of 30 cycles could as well be result of the overhead of calling the function and casting the 32-bit integer into a double. Finally, the `math.h` implementation will be able to handle overflows, underflows, exceptions and many other corner cases of IEEE-745, which our code can not. Also, it will result with increased precision in the calculation.

   Nevertheless, for this particular input data, and for this particular precision requirements, our tailored AVX implementation can win over the generic implementation, which is really important for writing fast numerical code.

```
================================================================
= Linux Perf Initializing
================================================================
Core 0  : online
Core 1  : online
Core 2  : online
Core 3  : online
Core 4  : offline
Core 5  : offline
Core 6  : offline
Core 7  : offline
Scheduling on core: 3

using libpfm4: 262144
Detected PMU models:
        [18, ix86arch, "Intel X86 architectural PMU"]
        [51, perf, "perf_events generic PMU"]
        [114, perf_raw, "perf_events raw PMU"]
        [200, skl, "Intel Skylake"]
Total events: 5739 available, 160 supported

Name        : CPU_CLK_THREAD_UNHALTED (Intel Skylake)
Description : Count core clock cycles whenever the clock
              signal on the specific core is running (not halted)
Code        : 0x3c

================================================================
= Using Linux Perf
================================================================
math_pow  : 207.068359
pow_scalar: 275.273438
pow_avx   : 173.605469

Validated!
```

Figure 2: Cycle count of `pow_avx` executed on Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, compiled with `gcc` version 4.8.5 20150623 (Red Hat 4.8.5-11), using `-O3 -march=core-avx2`, running on Red Hat Enterprise Linux Workstation release 7.3 (Maipo), `glibc` version 2.17