263-0007-00: Advanced Systems Lab

Assignment 1: 100 points
Due Date: Th, March 5th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2020/
Questions: fastcode@lists.inf.ethz.ch

Academic integrity:

All homeworks in this course are single-student homeworks. The work must be all your own. Do not copy any parts of any of the homeworks from anyone including the web. Do not look at other students code, papers, or exams. Do not make any parts of your homework available to anyone, and make sure no one can read your files. The university policies on academic integrity will be applied rigorously.

Submission instructions (read carefully):

- (Submission)
  Homework is submitted through the Moodle system https://moodle-app2.let.ethz.ch/course/view.php?id=12308. 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.
- (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 (e.g. for GCC use the flag "-O3") unless indicated otherwise.
- (Neatness)
  5% of the points in a homework are given for neatness.

Additional instructions:

- If you have an Intel processor, make sure to disable Turbo Boost in your computer to get accurate timing measurements.

Exercises:

1. (15 pts) Get to know your machine
   Determine and create a table for the following microarchitectural parameters of your computer:

   (a) Processor manufacturer, name, and number.
   (b) CPU base frequency.
   (c) CPU maximum frequency. Does your CPU support Turbo Boost or a similar technology?
   (d) Phase in the Intel's development model: Tick, Tock or Optimization. (if applicable)

   For one core and **without** using SIMD vector instructions determine:

   (d) Maximum theoretical floating point peak performance in flop/cycle.
   (e) Latency [cycles] and throughput [ops/cycle] for floating point division.
   (f) Latency [cycles] and throughput [ops/cycle] for floating point square root.
   (g) Latency [cycles] and throughput [ops/cycle] for floating point approximate reciprocal (`rcp`) instruction (if supported).

Notes:

- Intel calls throughput what is in reality the gap = 1/throughput.
- The manufacturer's website will contain information about the on-chip details. (e.g. Intel 64 and IA-32 Architectures Optimization Reference Manual).
- On Unix/Linux systems, typing `cat /proc/cpuinfo` in a shell will give you enough information about your CPU for you to be able to find an appropriate manual for it on the manufacturer's website (typically AMD or Intel).
- For Windows 7/10 "Control Panel/System and Security/System" will show you your CPU, for more info "CPU-Z" will give a very detailed report on the machine configuration.
- For Mac OS X there is "MacCPUID".
- Throughout this course, we will consider the FMA instruction as two floating point operations.

2. (25 pts) Matrix-vector multiplication

   In this exercise, we provide a C source file for multiplying an $n \times n$ matrix with a vector and a C header file to time the matrix-vector multiplication using different methods under Windows and Linux (for x86 compatible systems). Inspect and understand the code and do the following:

   (a) Using your computer, compile and run the code for $n = 400$. Compile using GCC with the highest level of optimization (use flag `-O3`). A modern compiler will automatically vectorize this very simple routine. Ensure you get consistent timings between timers and for at least two consecutive executions. Don't forget to disable Turbo Boost. (No need to answer anything here)

   (b) Determine the exact number of (floating point) additions and multiplications performed by the `compute()` function in `mvm.c`.

   (c) For all square matrices of sizes n between 200 and 4000, in increments of 200, create a performance plot with n on the x-axis and performance (in flops/cycle) on the y-axis. Create three series such that:

      i. The first series has all optimizations disabled: use flag `-O0`.

      ii. The second series has the major optimizations except for vectorization: use flags `-O3` and `-fno-tree-vectorize`.

      iii. The third series has all major optimizations enabled: use flags `-O3`, `-ffast-math` and `-march=native`.

   (d) Briefly discuss your plots and report the highest performance that you achieved.

3. (10 pts) Performance Analysis

   Assume that the elements of vectors x, y, u and z of length n are combined as follows:

$$z_i = z_i + u_i \cdot u_i \cdot u_i + x_i \cdot y_i \cdot z_i$$

   (a) Write a C/C++ `compute()` function that performs the computation described above on arrays of doubles. Save the file as combine.c(pp).

   (b) Within the same file create a benchmarking infrastructure based on the timing function that produces the most consistent results in Exercise 2.

   (c) Then, for all two-power sizes $n = 2^4, \ldots, 2^{23}$ create a performance plot with n on the x-axis (choose logarithmic scale) and performance (in flops/cycle) on the y-axis. Randomly initialize all arrays. For all n repeat your measurements 30 times reporting the median in your plot. Compile your code using GCC with flags `-O3` and `-march=native`.

   (d) Briefly explain performance variations in your plot.

   (e) Enroll and submit the code in Code Expert.

4. (30 pts) Cost analysis and bounds

   Consider the following algorithm that accurately multiplies a vector x with an $n \times n$ matrix A.

```
void accurate_mvm (double A[], double x[], double y[], int N)  {
    double sum_h, sum_l, Aij, Xj, th, tl, sh, sl, a, b;
    for(int i = 0; i < N; i++) {
        sum_h = 0.0;
        sum_l = 0.0;
        for(int j = 0; j < N; j++) {
            Aij  = A[n*i + j];
            Xj   = x[j];
            th   = Aij * Xj;
            tl   = fma(Aij, Xj, -th);
            sh   = th + sum_h;
            a    = sh - sum_h;
            b    = sh - a;
            sl   = (th - a) + (sum_h - b);
            sum_h = sh;
            sum_l += tl + sl;
        }
        y[i] = sum_h + sum_l;
    }
}
```

   We consider a Core i7 CPU based on a Haswell processor. As seen in the lecture, it offers FMA instructions (as part of AVX2). Assume that function `fma(a,b,c)` always is translated into an FMA instruction that computes `y = a * b + c`. Further, assume the bandwidths that are given in the additional material from the lecture: Abstracted Microarchitecture. Finally, assume that no optimization is performed that simplifies floating point arithmetic (i.e. `-ffast-math` flag is not used). Recall that we consider the FMA instruction as two floating point operations (an addition and a multiplication).

   (a) Define a suitable detailed floating point cost measure $C(n)$.

   (b) Compute the cost $C(n)$ of the function `accurate_mvm`.

   (c) Consider only one core without using vector instructions (i.e. using flag `-fno-tree-vectorize`) and determine a hard lower bound (not asymptotic) on the runtime (measured in cycles), based on:

      i. The throughput of the floating point operations. Assume that the only FMA instruction used is the one given by function `fma`. Be aware that the lower bound is also affected by the available ports offered for the computation (see lecture slides).

      ii. The throughput of the floating point operations where FMAs are used as much as possible to fuse an addition and a multiplication. For this exercise, it is also allowed to transform additions ($c := a + b$) into FMA instructions (of the form $c := 1.0 \cdot a + b$).

      iii. Loads, for the following two cases: All floating point data is L1-resident, and all floating point data is RAM-resident. Consider the best case scenario (peak bandwidth and ignore latency).

   (d) Determine an upper bound on the operational intensity. Assume empty caches and consider only reads but note: arrays that are only written are also read and the read should be included.

5. (20 pts) ILP analysis

   Consider the artificial computation below. The function operates on input arrays of length N.

```
double artcomp(double a[], double b[], double c[], int N) {
    double s = 0.0;
    for (int i = 0; i < N; i++) {
        s = (s + a[i] + b[i]) * c[i] + (a[i] * c[i] + s) * b[i];
    }
    return s;
}
```

   Make the same assumptions as in the previous exercises, i.e., consider a Haswell processor, only one core without using vector instructions (using flag `-fno-tree-vectorize`), and assume that no optimization is performed that simplifies floating point arithmetic (i.e. `-ffast-math` flag is not used). Thus, it is not allowed to apply associativity and distributivity laws to rearrange the computation. Determine hard lower bounds (not asymptotic) on the runtime (measured in cycles), based on:

(a) The latency, throughput and dependencies of the floating point operations. Assume that no FMA instruction is generated (i.e. `-mfma` flag is not used). Be aware that the lower bound is also affected by the available ports offered for the computation (see lecture slides).

(b) The latency, throughput and dependencies of the floating point operations where FMAs are used as much as possible to fuse an addition and a multiplication (i.e. `-mfma` flag is enabled). For this exercise, it is **not** allowed to transform additions ($c := a + b$) into FMA instructions (of the form $c := 1.0 \cdot a + b$).

Note: It may be useful to draw the Directed Acyclic Graph (DAG) of the computation.

How to disable Intel Turbo Boost

Intel Turbo Boost is a technology implemented by Intel in certain versions of its processors that enables the processor to run above its base operating frequency via dynamic control of the processor's clock rate. It is activated when the operating system requests the highest performance state of the processor.

BIOS

Intel Turbo Boost Technology is typically enabled by default. You can only disable and enable the technology through a switch in the BIOS. No other user controllable settings are available. Once enabled, Intel Turbo Boost Technology works automatically under operating system control. When access to BIOS is not available, few workarounds are possible:

Linux

Linux does not provide interface to disable Turbo Boost. One alternative, that works, is disabling Turbo Boost by writing into MSR registers. Assuming 2 cores, the following should work:

```
wrmsr -p0 0x1a0 0x4000850089
wrmsr -p1 0x1a0 0x4000850089
```

To enable it:

```
wrmsr -p0 0x1a0 0x850089
wrmsr -p1 0x1a0 0x850089
```

This method has been criticized here and, here stating that the OS can circumvent the MSR value, using opportunistic strategy. But so far in our tests, we have observed that Linux conforms to the MSR value. An alternative method would be to use `cpupower`, as explained in the ArchLinux Wiki, as well as the the intel_pstate driver. Unfortunately, we can not confirm deterministic behavior across different kernel versions with this method.

Mac OS X

Disabling Turbo Boost in OS X can be done easily with the Turbo Boost Switcher for OS X. Note that the change is not persistent after restart. The method also writes to the MSR register, and shares the same weaknesses as the Linux approach.

Windows

Windows does not provide any functionality to disable Intel Turbo Boost. The only effective way of disabling is using the BIOS. On some Intel machines however, it is possible to fix the CPU multiplier such that the resulting frequency corresponds to the nominal frequency of the CPU. ThrottleStop provides this functionality with a convenient GUI. “Disable Turbo” will effectively fix the frequency such that it corresponds to a behaviour of a CPU with disabled Turbo Boost.