263-0007-00: Advanced Systems Lab

Assignment 1: 100 points
Due Date: Th, March 9th, 17:00
https://acl.inf.ethz.ch/teaching/fastcode/2023/
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 and through Code Expert for coding exercises. The enrollment link for Code Expert is https://expert.ethz.ch/enroll/SS23/asl.
- (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**.
- (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 points in this 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, number and microarchitecture (e.g. Haswell, Skylake, etc).

   (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)

   Intel's processors offer two different floating-point instruction sets, namely x87 and SSE/SSE2, that can perform scalar floating-point operations. For example, a floating-point division can be performed using either FDIV (from x87) or DIVSD (from SSE2) assembly instructions. The x87 instruction set, however, is becoming deprecated but is still supported for backward compatibility.

   (e) Name three differences between x87 and SSE2.

For one core and **without** using SIMD vector instructions, determine the following about your machine. In (g)-(h), make sure to use the correct floating-point instruction (not the one from x87 in case you have an Intel processor) and provide the reference where you found the latency and throughput information.

(f) Maximum theoretical floating-point peak performance in flops/cycle.

(g) Latency [cycles], throughput [ops/cycle] and instruction name for both double- and single-precision floating-point multiplication.

(h) Latency [cycles], throughput [ops/cycle] and instruction name for both double- and single-precision square root.

(i) Latency [cycles], throughput [ops/cycle] and instruction name for double-precision ceiling operation, i.e., the operation that rounds a floating-point number up to an integer-valued floating-point.

Notes:

- Latency and throughput information for Intel's and AMD's processors can be found in Agner Fog's instruction tables, uops and Intel's optimization manual. See Dougall Johnson's documentation for Apple's M1 processors.
- 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 macOS there is `sysctl machdep.cpu.brand_string`.
- Throughout this course, we will consider the FMA instruction as two floating-point operations.

2. (20 pts) Matrix multiplication

In this exercise, we provide a C source file for multiplying an $n \times n$ matrix with its transpose and a C header file that allows to read the time stamp counter (TSC) of the processor for x86 compatible systems. The code uses different timers available to time the matrix multiplication. Note that if you have an Apple M1/M2 processor, you can still access some of the timers available so you can still complete the homework. Inspect and understand the code and do the following:

(a) Using your computer, compile and run the code. Compile with the highest level of optimization provided by your compiler (with GCC, compile with the 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) Inspect the `compute()` function in `mmm.c` and answer the following:

i. Determine the exact number of floating-point additions and multiplications that it performs.

ii. Determine an upper bound on its operational intensity (consider only reads and assume empty caches).

(c) For all square matrices of sizes n between 100 and 1500, in increments of 100, create a performance plot with n on the x-axis and performance (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`. If you are using the clang compiler, also add `-fno-slp-vectorize` flag to disable vectorization.

iii. The third series has all major optimizations enabled including vectorization: use flags `-O3`, `-ffast-math` and `-march=native`. If you are using an Apple M1 processor and your compiler doesn't support `-march=native` you can use `-mcpu=apple-m1` instead.

(d) Discuss performance variations of your plots and report the highest performance that you achieved.

3. (25 pts) Performance analysis and bounds

Assume that vectors u, w, x, y and z of length n are implemented using double precision floating-point and combined as follows:

$$z_i = u_i \cdot u_i \cdot u_i + z_i \cdot \max(x_i - y_i, u_i - w_i)$$

We consider a Core i7 CPU with a Skylake microarchitecture. As seen in the lecture, it offers FMA instructions (as part of AVX2). Recall that we consider cost of the FMA instruction as two floating-point operations (an addition and a multiplication). Assume the bandwidths that are given in the additional material from the lecture: Abstracted Microarchitecture. Assume that no optimization is performed that simplifies floating-point arithmetic (i.e. `-ffast-math` flag is not used) and that the max function is translated to a `maxsd` instruction by the compiler. Answer the following and justify your answers.

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

(b) Compute the cost $C(n)$ of the computation.

(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 no FMA instructions are used. 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 to fuse an addition and a multiplication (i.e. `-mfma` flag is enabled).

iii. The throughput of data reads, for the following two cases: All floating-point data is L3-resident, and all floating-point data is RAM-resident. Consider the best case scenario (peak bandwidth and ignore latency). Note that arrays that are only written are also read and this read should be included.

(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 this read should be included.

4. (25 pts) Basic optimization

Consider the following function:

```c
void comp(double *x, double *y, int n) {
    double s = 0.0;
    for (int i = 0; i < n; i++) {
        s = (s + x[i]*x[i]) + y[i]*y[i]*y[i];
    }
    x[0] = s;
}
```

(a) Create a benchmarking infrastructure based on the timing function that produces the most consistent results in Exercise 2 and for all two-power sizes $n = 2^4, \ldots, 2^{23}$ create a performance plot for the function `comp` 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 with flags `-O3 -mfma`[1] `-fno-tree-vectorize`. If you are using clang, add also the `-fno-slp-vectorize` and `-ffp-contract=fast` flags.

[1]For Apple M1/M2 processors, the flag `-mfma` may not be supported. If this is the case, use instead `-mcpu=apple-m1` or `-march=native`.

(b) Considering the latency and throughput information of floating-point operations in your machine, and the dependencies in `comp`, derive an upper bound on the performance (flops/cycles) of `comp` when using the specified flags in (a), i.e., when FMA instructions are enabled (`-mfma`) but vectorization is disabled (`-fno-tree-vectorize`).

(c) Perform optimizations that increase the ILP of function `comp` to improve its runtime. It is not allowed to use vector instructions. Add the performance to the previous plot (so one plot with two series in total for (a) and (c)). Compile your code with the same flags as before.

(d) Discuss performance variations of your plot and report the highest performance that you achieved. Also discuss the optimizations that you performed to increase the ILP.

(e) Enroll and submit the code of your optimized function in Code Expert. Carefully read and follow the instructions given in Code Expert to submit your code.

5. (10 pts) ILP analysis

Consider the following computations:

```c
#include <math.h>

double artcomp1(double a, double b, double c, double d) {
    double r;
    r = (a*a + b*b) - (c*c + d*d);
    return r;
}

double artcomp2(double a, double b, double c, double d) {
    double r;
    r = ceil(a)*b - ceil(c)*d;
    return r;
}
```

Make the same assumptions as in Exercise 3, i.e., consider a Skylake 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) for the following cases, based on the latency, throughput and dependencies of the floating-point operations only. 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). It may be useful to draw the dependency graph of the computation. Justify your answers.

(a) Determine a hard lower bound on the runtime for `artcomp1`.

(b) Determine a hard lower bound on the runtime for `artcomp2`. Assume that the compiler transforms the `ceil` function to the respective assembly instruction performing this operation (so no function call to the math library occurs).

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 an 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.