Exercises for Architectures of Supercomputers

2nd Exercise, 30./31.10.2019

Johannes Hofmann

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Exercise 2: Pipelining

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

- In the lecture, you learned that the instruction cycle can be pipelined to increase instruction throughput
- The same concept can be applied to execution units, if operations take longer than one cycle to execute

Exercise 2: Pipelining

- In the lecture, you learned that the instruction cycle can be pipelined to increase instruction throughput
- The same concept can be applied to execution units, if operations take longer than one cycle to execute

```
123456.7 + 101.7654 =
```

Familiar mathematical repre-
sentationof numbers

Exercise 2: Pipelining

- In the lecture, you learned that the instruction cycle can be pipelined to increase instruction throughput
- The same concept can be applied to execution units, if operations take longer than one cycle to execute

```
123456.7 + 101.7654 = (1.234567 × 10^5) + (1.017654 × 10^2)
```

Floating-point number representation
(always one digit left of the decimal point)

Exercise 2: Pipelining

- In the lecture, you learned that the instruction cycle can be pipelined to increase instruction throughput
- The same concept can be applied to execution units, if operations take longer than one cycle to execute

```
123456.7 + 101.7654 = (1.234567 × 10^5) + (1.017654 × 10^2)
                    = (1.234567 × 10^5) + (0.001017654 × 10^5)
```

First step of floating-point addition:
Shift mantissa to make exponents match

Exercise 2: Pipelining

- In the lecture, you learned that the instruction cycle can be pipelined to increase instruction throughput
- The same concept can be applied to execution units, if operations take longer than one cycle to execute

```
123456.7 + 101.7654 = (1.234567 × 10^5) + (1.017654 × 10^2)
                    = (1.234567 × 10^5) + (0.001017654 × 10^5)
                    = (1.234567 + 0.001017654) × 10^5
```

Second step of floating-point addition:
Perform the actual addition

Exercise 2: Pipelining

- In the lecture, you learned that the instruction cycle can be pipelined to increase instruction throughput
- The same concept can be applied to execution units, if operations take longer than one cycle to execute

```
123456.7 + 101.7654 = (1.234567 × 10^5) + (1.017654 × 10^2)
                    = (1.234567 × 10^5) + (0.001017654 × 10^5)
                    = (1.234567 + 0.001017654) × 10^5
                    = 1.235584654 × 10^5
```

Third step of floating-point addition:
Rounding and normalization of result

Exercise 2: Pipelining

- In the lecture, you learned that the instruction cycle can be pipelined to increase instruction throughput
- The same concept can be applied to execution units, if operations take longer than one cycle to execute

```
123456.7 + 101.7654 = (1.234567 × 10^5) + (1.017654 × 10^2)
                    = (1.234567 × 10^5) + (0.001017654 × 10^5)
                    = (1.234567 + 0.001017654) × 10^5
                    = 1.235584654 × 10^5
```

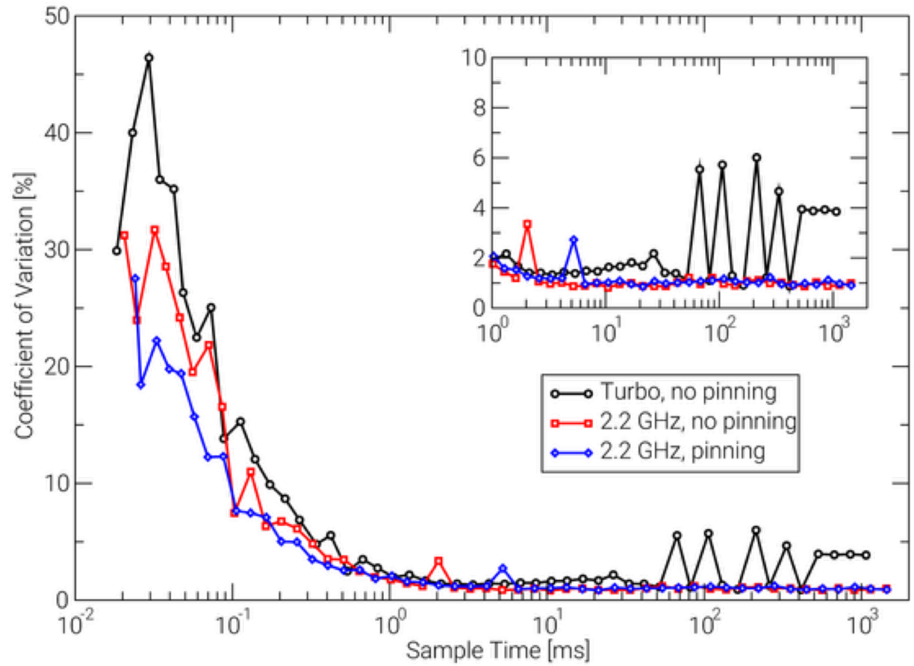
- Rationale: Use hardware implementing different stages simultaneously to increase **throughput** (number of operations over time)
- The **latency** (execution time of one operation) is unchanged

Exercise 2: Vector sum

- Continue with your code from last week
- Implement a function `double vec_sum(double *A, int N)`, that adds up all elements of an array `A` of length `N` and returns the result
  - As before, make sure to implement the function in a separate file
- In your existing code, run the `vec_sum` function on the array after it is initialized by the `init` function
  - Change the `init` function to initialize your array elements with 1.0
    - Allows easy verification of the correctness of your `vec_sum` implementation
  - Instead of measuring the runtime of the `init` function, this week you should measure the runtime of the `vec_sum` function
  - Instead of bandwidth, the performance metric for the `vec_sum` function should be floating-point operations per second (Flop/s)
    - To derive the performance the number of floating-point operations carried out in the `vec_sum` function is divided by the function's runtime
  - Make sure the runtime of your `vec_sum` function is at least 0.1 seconds
    - Call the function multiple times if necessary (see next slide)

Exercise 2: Minimum runtime

- Why enfore a minimum runtime of 100ms?
  - Various effects can bias performance when using small sample times
    - OS (scheduler, interrupts, ...)
    - Overhead (startup, function, ...)
- How to enforce minimum runtime?



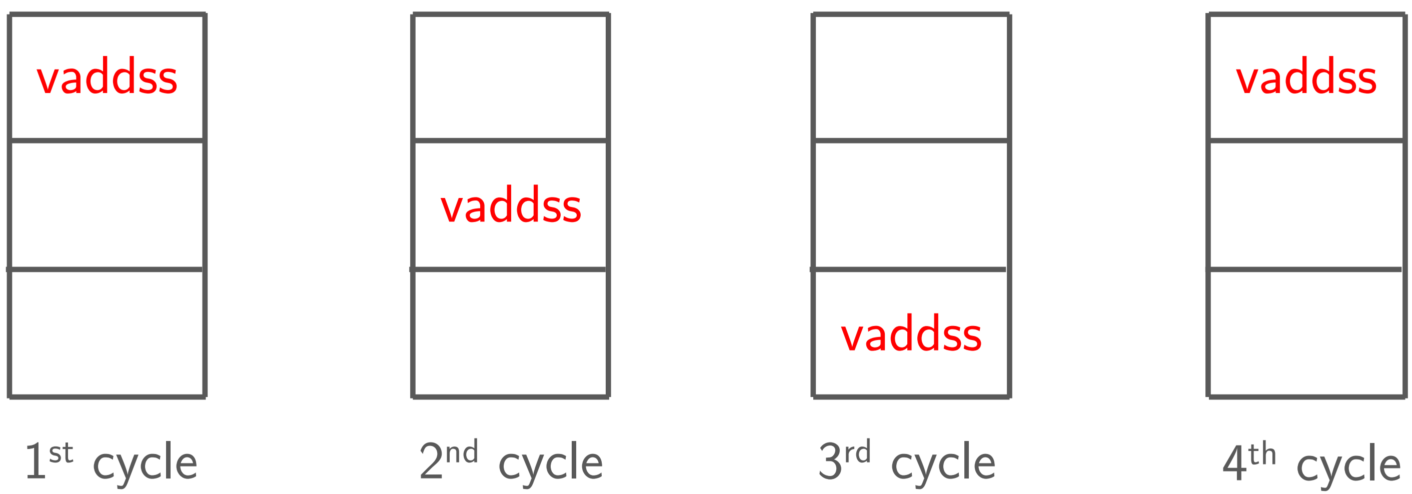

```
for (R=1; runtime<0.1; R*=2) {
  start = get_time();
  for (i=0; i<R; ++i)
    // Measurement loop is executed R times
    vec_sum(A, N);
  stop = get_time();
  runtime = stop − start;
}
R = R / 2;  // update expr executed before exit cond. s evaluated
```

Pipelining and unrolling

```
#pragma novector
#pragma nounroll
for (i=0; i<N; ++i)
        sum += A[i];
```

compiler

```
..B1.5:
vaddss     xmm0, xmm0, [rdi+rax*4]
inc        rax
cmp        rax, rsi
jl         ..B1.5
```



Exercise 2: Unrolling to improve performance

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

- In this exercise, you will investigate the impact of loop-unrolling on performance
- Start with an implementation of the `vec_sum` function that does not use unrolling
  - To prevent the compiler from automatically unrolling your high-level C code, use `#pragma nounroll` in addition to `#pragma novector` in front of your for-loop in your `vec_sum` function
- Next, implement 2-, 3-, 4-, and 8-way unrolling versions of `vec_sum` function
  - Implement them in separate files and functions
    - E.g., `vec_sum2.c` containing the two-way unrolled `vec_sum2` and so on…
  - Take care to correctly address the remainder loop
  - See blackboard…

Exercise 2: Measurements

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

- Measure the performance of your different vector-sum implementations

- Use a data-set size of 23kB
  - large enough to be non-trivial
  - small enough to fit into the L1 cache

- Try to find an explanation for the observed performance
  - The CPU cores are clocked with a frequency of 2.2 GHz
  - According to the *Intel Optimization Reference Manual* [1] the `addss` instruction has a latency of three clock cycles on the Ivy Bridge microarchitecture

[1] http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html, cf. Tab C-15