Exercises for Architectures of Supercomputers

8th Exercise, 15./16.1.2020

Johannes Hofmann

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Previously…

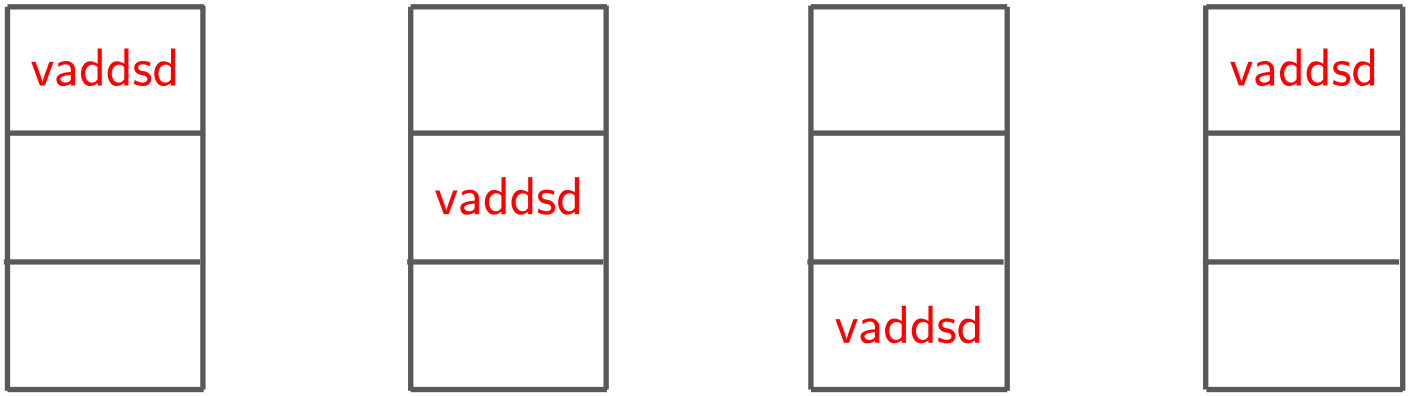
- Let’s revisit exercise 2…

Pipelining and unrolling

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

compiler

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

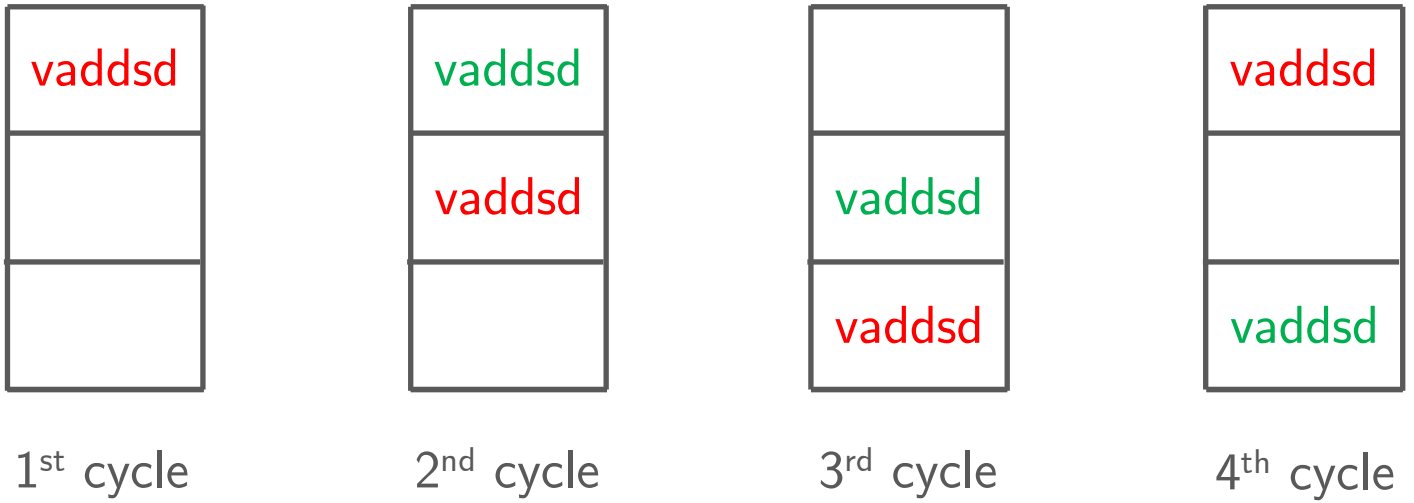


Pipelining and unrolling

```
#pragma novector
#pragma nounroll
for (i=0; i<N; i+=2) {
    sum0 += A[i];
    sum1 += A[i+1];
}
```

compiler

```
..B1.5:
vaddsd    xmm1, xmm1, [rdi+rax*4]
vaddsd    xmm0, xmm0, [4+rdi+rax*4]
add       rax, 2
cmp       rax, rsi
jl        ..B1.5
```

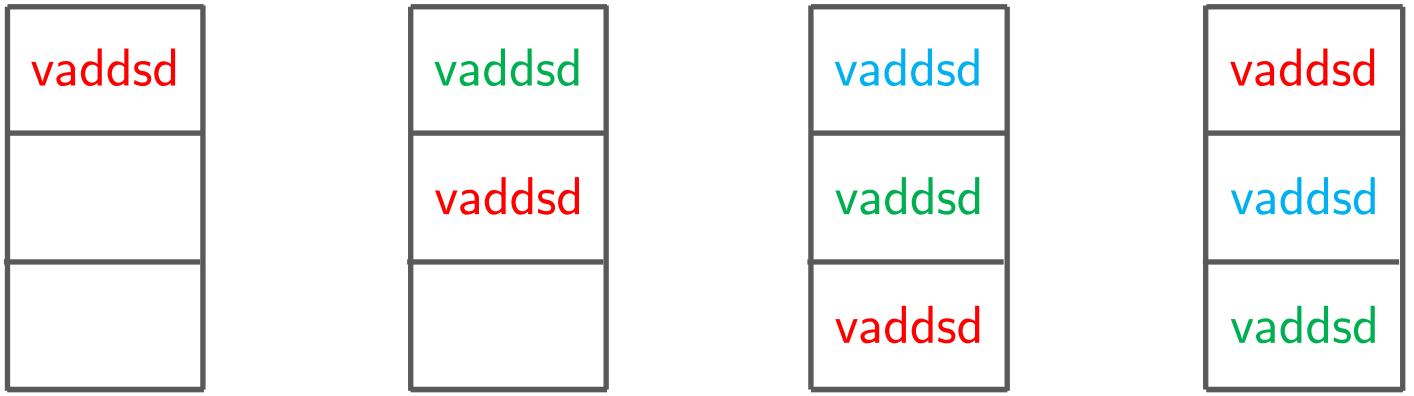


Pipelining and unrolling

```
#pragma novector
#pragma nounroll
for (i=0; i<N; i+=3) {
    sum0 += A[i];
    sum1 += A[i+1];
    sum2 += A[i+2];
}
```

compiler

```
..B1.5:
vaddsd    xmm2, xmm2, [rdi+rax*4]
vaddsd    xmm1, xmm1, [4+rdi+rax*4]
vaddsd    xmm0, xmm0, [8+rdi+rax*4]
add       rax, 3
cmp       rax, rsi
jl        ..B1.5
```



Implications on Performance

Using the CPUs from ***emmy*** as example (CPU core clock: 2.2 GHz)

- No unrolling
  - One add (= one floating-point operation) is completed every three cycles
  - 2.2 GHz x 1 Flop / 3 cycles → 733 MFlop/s
- Using two-way unrolling
  - Two adds (flops) are completed every three cycles
  - 2.2 GHz x 2 Flops / 3 cycles → 1467 MFlop/s
- Using three-way or more unrolling
  - Three adds (flops) are competed every three cycles
  - 2.2 GHz x 3 Flops / 3 cycles = 2200 MFlop/s

Exercise 8 (Overview)

- Use SIMD vector instructions to improve performance
  - Use the vector add instruction (`vaddpd`) Instead of a scalar add instructions (`vaddsd`) to carry out four double-precision add operations with a single instruction
  - Start with the naïve code and **leave the novector and nounroll pragmas in the code!**

```
double vec_sum(double *A, int N)
  #pragma novector
  #pragma nounroll
  int i;
  for (i=0; i<N; ++i)
    sum += A[i];
  return sum;
}
```

  - Use intrinsics to make the compiler generate SIMD instructions

Exercise 8 (Intrinsics)

- Intrinsics are functions that map to a particular instruction
  - Example: `__m256d _mm256_add_pd (__m256d a, __m256d b)`
    - The `_mm256_add_pd` intrinsic function takes to two 256-bit registers as input, performs a component-wise addition, and stores the result in a 256-bit register. The corresponding instruction is `vaddpd`.
    - Example:

```
__m256d input1 = ...;
__m256d input2 = ...;
__m256d result = _mm256_add_pd(input1, input2);
```

  - You can find a list of all intrinsics here: https://software.intel.com/sites/landingpage/IntrinsicsGuide
    - See demonstration on how to use this website

Exercise 8.1 (SIMD vectorization)

- Workflow outline
  - Before the loop, initialize a vector register for the sum
    - All four DP FP numbers should be initialized with `0.0`. Search the intrinsics guide for '`set*pd`' to find an appropriate instruction
  - In the loop body, use an (unaligned) AVX load instruction to load four consecutive double-precision floating-point numbers from memory into a vector register
    - Search the intrinsics guide for '`load*pd`' to find an appropriate instruction
  - Then, add the contents of the vector register containing the loaded values to the vector register containing the current sums
  - After the loop, find a way to do a horizontal sum of the contents of the vector register
    - Use a combination of `hadd` and `insertf` (look at intrinsics guide for details)
    - Store the results (look for '`store*pd`' in the intrinsics guide) of the vector register to memory (e.g., to `double tmp[4]`), and sum up the values sequentially (e.g., `return tmp[0]+tmp[1]+...;`)
    - Find your own solution
  - Make sure to adjust the loop increment

Exercise 8.1 (SIMD vectorization)

- Measure the performance [in MFlop/s] of your SIMD-vectorized code for the 64 different data-set sizes specified in the `input_sizes.txt` file provided in StudOn
  - Note that the sizes in the file correspond to kilobytes
- Visualize your measurements in a graph
  - Display the data-set size on a logarithmic *x*-axis
  - Show the measured performance on the *y*-axis

Exercise 8.2 (SIMD vect. and unrolling)

Apply unrolling to your loop to increase performance further

- When using SIMD instructions, the same issues with respect to data hazards apply as when using scalar instructions
  - One SIMD sum register + three cycles latency = Losing one third of performance
- Apply four-way unrolling to your code
  - Use four vector registers to hold partial sums (for a total of sixteen double-precision sums)
  - Carry out four vector loads and four vector adds in the loop
  - After the loop, apply sum-reduction on four vector registers
  - After that, calculate horizontal sum of resulting vector register to get scalar result

Exercise 8.2 (SIMD vect. and unrolling)

- Measure the performance [in MFlop/s] of your SIMD-vectorized and unrolled code for data-set sizes in `input_sizes.txt`
- Add the results to the existing plot from exercise 8.1

Exercise 8.3

- Next, go back to the naïve C code and remove the two `#pragma`s
- Compile the code with `-O3` and `-qopt-report=3`
  - `-qopt-report=3` will generate an optimization report. Look at it and find out whether your code was successfully vectorized; in addition, you can also look at the resulting assembly (can be generated with the `-S` flag) and check for packed instructions
  - Measure the performance of the compiler-generated code and include it in the existing plot
- Now compile the code with `-O3`, `-qopt-report=3` and **`-xHost`**
  - Measure the performance of the compiler-generated code and include it in the existing plot
  - What changed? How does it influence performance?