NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

Programming Techniques for Supercomputers Tutorial

Erlangen National High Performance Computing Center

Department of Computer Science

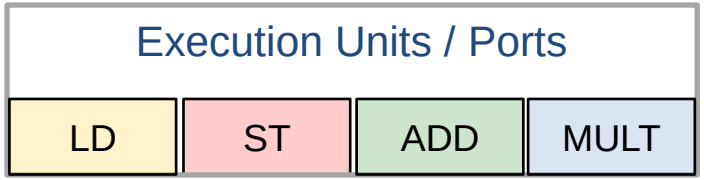
FAU Erlangen-Nürnberg

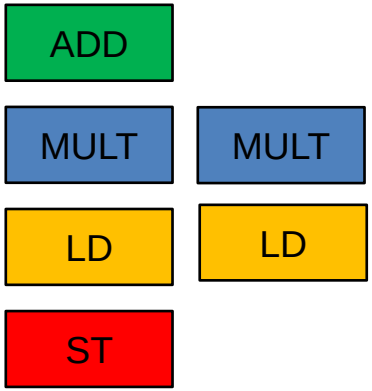
Sommersemester 2024

Assignment 1 – Task 1

Pipelines

```
double a[...],b[...];
double s=0.0,t=1.234;
// a[] and b[] contain sensible data
for(int i=0; i<N; ++i) {
    s += a[i]*a[i];
    b[i] *= t;
}
```



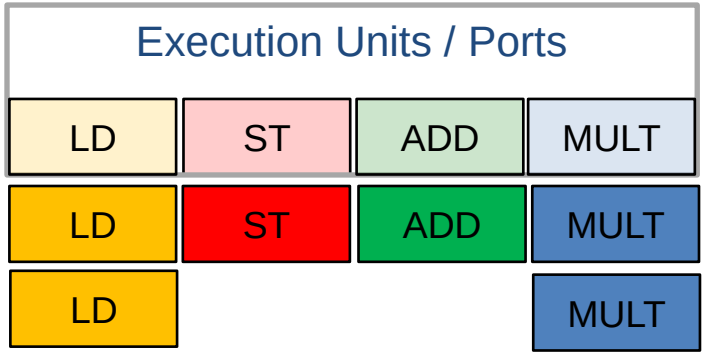


- 1 ADD
- 2 MULTs
- 2 LOADs
- 1 STORE

Assignment 1 – Task 1

Pipelines

```
double a[...],b[...];
double s=0.0,t=1.234;
// a[] and b[] contain sensible data
for(int i=0; i<N; ++i) {
    s += a[i]*a[i];
    b[i] *= t;
}
```



- 1 ADD
- 2 MULTs
- 2 LOADs
- 1 STORE

Does this mean we can do 1 iter. per 2 cy (i.e., 3 flops in 2 cy)?

NO

Remember: Pipelining and dependencies from lecture (slide set 3a)

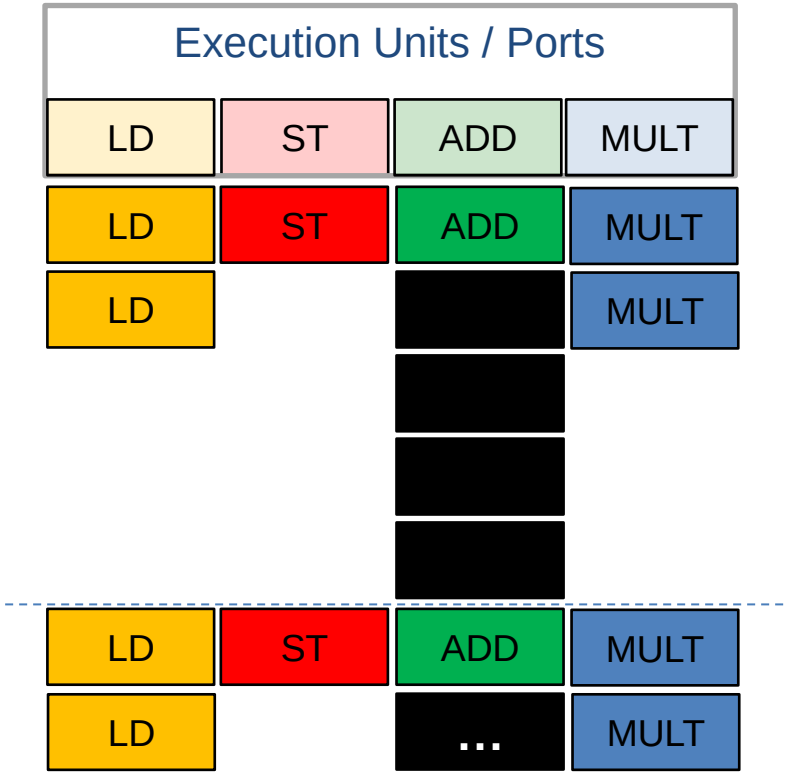
Assignment 1 – Task 1

Pipelines without MVE

```
double a[...],b[...];
double s=0.0,t=1.234;
// a[] and b[] contain sensible data
for(int i=0; i<N; ++i) {
    s += a[i]*a[i];
    b[i] *= t;
}
```

Bottleneck: ADD pipeline stall (latency)

➔ P_{max} = 1 iter/5 cy = 3 flops/ 5 cy
= 0.6 flops/cy

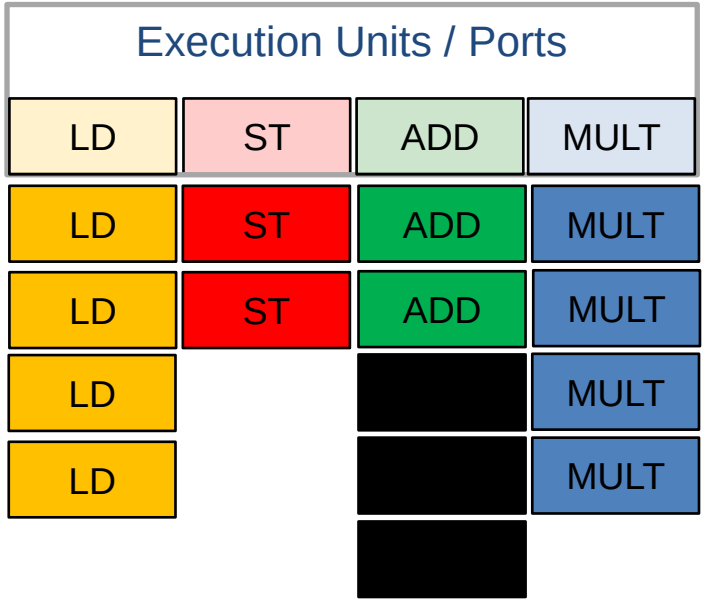


Assignment 1 – Task 1

Pipelines and MVE

2 times MVE

```
double a[...],b[...];
double s=0.0,t=1.234;
double s0=0.0, s1=0.0;
// a[] and b[] contain sensible data
for(int i=0; i<N; i=i+2) {
    s0 += a[i]*a[i];
    s1 += a[i+1]*a[i+1];
    b[i] *= t;
    b[i+1] *= t;
}
s = s0+s1;
```



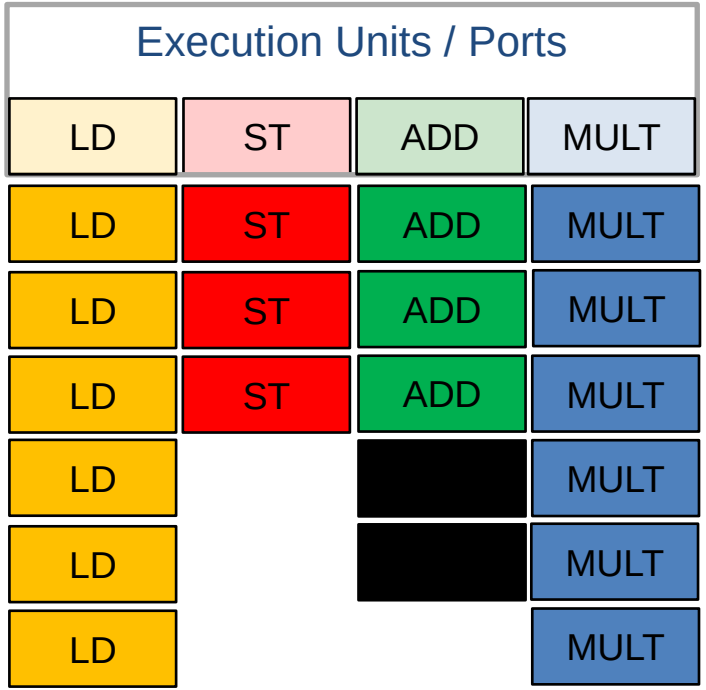
Still bottleneck: ADD pipeline stall (latency)

➔ P_{max} = 2 iter/5 cy = 6 flops/ 5 cy = 1.2 flops/cy

Assignment 1 – Task 1

Pipelines and MVE

3 times MVE



Bottleneck: LOAD & MULT pipeline

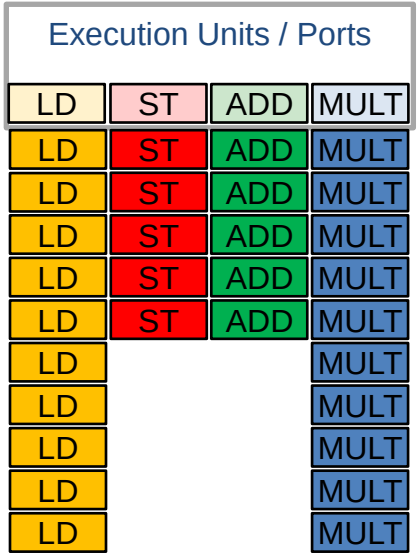
➔ P_{max} = 3 iter/6 cy = 9 flops/ 6 cy = 1.5 flops/cy

Assignment 1 – Task 1

Pipelines and MVE

5 times MVE will fill the ADD pipeline.
But, P_{max} reaches its limit at 3 times MVE, due to new LD & MULT bottleneck.

➔ Best possible P_{max} = 1.5 flops/cy



Assignment 1 – Task 2

Peak Performance

56 (cores)
x 16 (elements/register)
x 2 (flops/FMA)
x 2 (FMA/cy)
x 2.2 (Gcy/s) = 7.8848 Tflop/s

Assignment 1 – Task 3

Square Roots Reloaded

a)

```
# PI=3.141592653589793
$ srun --cpu-freq=2000000-2000000 ./a.out.scalar
Pi = 3.141592653589688 in 3.080 s -> 6.16 cy/it
$ srun --cpu-freq=2000000-2000000 ./a.out.SSE
Pi = 3.141592653589613 in 1.508 s -> 3.02 cy/it
$ srun --cpu-freq=2000000-2000000 ./a.out.AVX2
Pi = 3.141592653589493 in 1.510 s -> 3.02 cy/it
$ srun --cpu-freq=2000000-2000000 ./a.out.AVX512
Pi = 3.141592653589807 in 1.509 s -> 3.02 cy/it
```

b)

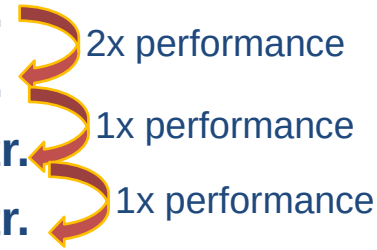
- 1 – 8 summation variables → **different order of accumulation**
- maybe MVE on top of SIMD vectorization
- **different number of partial sums**

Assignment 1 – Task 3

Square Roots Reloaded

c)

- Scalar : **3.08s** → `sqrt` instruction inverse TP = **6 cy/instr.**
- SSE: **1.51s** → `sqrt` instruction inverse TP = **6 cy/instr.**
- AVX: **1.51s** → `sqrt` instruction inverse TP = **12 cy/instr.**
- AVX-512: **1.51s** → `sqrt` instruction inverse TP = **24 cy/instr.**



- to save transistors (?), only execute SQRTs ≥ 16B in packs of 128b/SSE width
  ➔ only two SIMD lanes available for vector SQRT