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 2026

# Assignment 4 – Task 1

## Optimizing for code balance

a)

```
float a[],b[],c[],z[];

for(i=0; i<N; ++i) {
  a[i] += b[i] * c[i];
}
for(i=0; i<N; i+=2) {
  z[i] = a[i] * 0.5f;
}
```

Loop fusion

```
float a[],b[],c[],z[];

for(i=0; i<N; ++i) {
  a[i] += b[i] * c[i];
  if (!(i & 1)) {
    z[i] = a[i] * 0.5f;
  }
}
```

Bitwise check if i is even

$$B_c = \frac{16+12}{2+0.5}\,\frac{\mathrm{B}}{\mathrm{Flop}} = \mathbf{11.2\ B/Flop}$$

Half a FLOP for stride 2 loop

$$B_c = \frac{16+8}{2.5}\,\frac{\mathrm{B}}{\mathrm{Flop}} = \mathbf{9.6\ B/Flop}$$

# Assignment 4 – Task 1

## Optimizing for code balance

b)

- L1-L2: 64 B/cy half-duplex → **1 cy/CL**
- L2-L3: 16 B/cy half-duplex → **4 cy/CL**
- Mem BW: 110 GB/s, half-duplex → $\frac{64\text{ B} * 2.4\text{ Gcy/s}}{110\text{ GB/s}} = \mathbf{1.4\ cy/CL}$
- 2.4 GHz clock freq

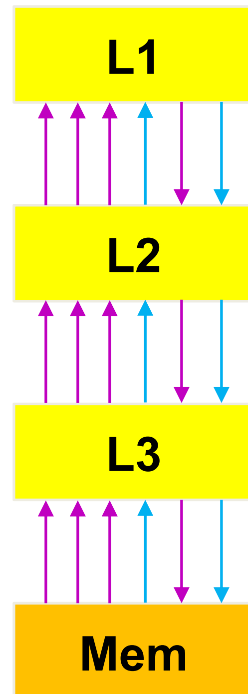
```
float a[],b[],c[],z[];

for(i=0; i<N; ++i) {
  a[i] += b[i] * c[i];
  if(!(i & 1)) {
    z[i] = a[i] * 0.5f;
  }
}
```

**L1-L2:** **6 cy**

**L2-L3:** **24 cy**

**Memory:** **8.4 cy**



# Assignment 4 – Task 2

## Code balance

a) Single Precision `s = s + a[i];` $B_c = 4\ [B/F]$

b) Double Precision `s = s + a[i]*a[i];` $B_c = 4\ [B/F]$

c) Single Precision `a[i][j] = a[i][j]+ s*c[i][j];` $B_c = 6\ [B/F]$

d) INT32 Ops `a[i] = s*b[i];` $B_c = 12\ [B/F]$

e) Double Precision, int32 `x[i] = s*v[index[i]];`

- Case 1:
  - `index[i]=const`, hence no extra traffic from `v[]` ⇒ $B_c = 20[B]/1[F] = \mathbf{20}\ [B/F]$
- Case 2:
  - `index[i]=i`, means that `v[]` incurs 8 B ⇒ $B_c = 28[B]/1[F] = \mathbf{28}\ [B/F]$
- Case 3:
  - `index[i]=rand()`, `v[]` incurs one 64 B CL ⇒ $B_c = 84[B]/1[F] = \mathbf{84}\ [B/F]$

# Assignment 4 – Task 3

## Square Roots Reloaded

a)

```
# PI=3.141592653589793
$ srun --cpu-freq=2000000-2000000:performance ./a.out.scalar
Result=3.141592653590036 in 2.061e+00 s -> 6.14 cy/it
$ srun --cpu-freq=2000000-2000000:performance ./a.out.SSE
Result=3.141592653589690 in 1.011e+00 s -> 3.01 cy/it
$ srun --cpu-freq=2000000-2000000:performance ./a.out.AVX2
Result=3.141592653589769 in 1.012e+00 s -> 3.01 cy/it
$ srun --cpu-freq=2000000-2000000:performance ./a.out.AVX512
Result=3.141592653589739 in 1.011e+00 s -> 3.02 cy/it
```

b) **Observation 1**

- Using 1–8 summation variables **changes the order of accumulation**
- Possible Modulo Variable Expansion (**MVE**) on top of SIMD vectorization
- **Different number of partial sums** generated by the compiler

# Assignment 4 – Task 3

## Square Roots Reloaded

b) **Observation 2**

- Scalar: **2.06s** → **`sqrt`** inverse throughput = **~6 cy/instr.**
- SSE: **1.01s** → **`sqrt`** inverse throughput = **~6 cy/instr.**
- AVX: **1.01s** → **`sqrt`** inverse throughput = **~12 cy/instr.**
- AVX-512: **1.01s** → **`sqrt`** inverse throughput = **~24 cy/instr.**

2x performance

1x performance

1x performance

- To save transistors (?), vector SQRT instructions wider than 128 bits are internally executed as multiple 128-bit (“SSE-width”) operations
  ➔ Effectively only two SIMD lanes handle vector SQRT