

SIMD-Intrinsics

- ▶ Neuer Datentyp `__m128`
- ▶ SSE-Instruktion → Intrinsic
 - ▶ `addps xmm, xmm → __m128 _mm_add_ps (__m128 a, __m128 b)`
 - ▶ `mulss xmm, xmm → __m128 _mm_mul_ss (__m128 a, __m128 b)`
- ▶ `mov*s`
 - ▶ Load (aps) → `__m128 _mm_load_ps(float const* mem_addr)`
 - ▶ Store (aps) → `void _mm_store_ps(float* mem_addr, __m128 a)`

Quiz: Load

Sei `b` vom Typ `float`. Wie wird `x` korrekt initialisiert?

☐

```
__m128 x = b;
```

☐

```
__m128 x = _mm_load_ss (b);
```

☐

```
__m128 x = _mm_load_ss (&b);
```

Quiz: Register

Wie kann das Register xmm0 auf 0 gesetzt werden?

☐

`float z = 0; xmm0 = _mm_load1_ps(&z);`

☐

`_mm_xor_ps(xmm0, xmm0);`

☐

Nicht wie oben, da in C von Registern abstrahiert wird

Codebeispiel

► $y \leftarrow \alpha * x + y$

```
1 #include <immintrin.h>
2
3 void saxpy(long n, float alpha, float* x, float* y){
4     __m128 valpha = _mm_load1_ps(&alpha);
5     for(size_t i = 0; i < n; i += 4){
6         __m128 vx = _mm_loadu_ps(x+i);
7         __m128 vy = _mm_loadu_ps(y+i);
8
9         vy = _mm_add_ps(_mm_mul_ps(valpha, vx), vy);
10
11         _mm_storeu_ps(y+i, vy);
12     }
13     // ...
14 }
```

Codebeispiel

► $y \leftarrow \alpha * x + y$

```
1 #include <immintrin.h>
2
3 void saxpy(long n, float alpha, float* x, float* y){
4     // ...
5     for(size_t i = (n-(n%4)); i < n; i++){
6         y[i] = alpha*x[i] + y[i];
7     }
8 }
```

Quiz: Alignment

Was passiert wenn `loadu_ps` durch `load_ps` ersetzt wird?

☐

Das Programm wird schneller

☐

Das Programm stürzt ab

☐

Der Compiler erzwingt 16 Byte Alignment für `x` und `y`

Andere Datentypen

- ▶ `__m128, __m128d, __m128i`
- ▶ `addpd xmm, xmm → __m128d _mm_add_pd (__m128d a, __m128d b)`
- ▶ `paddb xmm, xmm → __m128i _mm_add_epi8 (__m128i a, __m128i b)`

Quiz: Integer

Was ist die korrekte Signatur des Intrinsics für psubq auf xmm-Registern?

☐

`__m128i _mm_sub_epi64 (__m128i a, __m128i b)`

☐

`__m64 _mm_sub_si64 (__m64 a, __m64 b)`

☐

`__m128i _mm_sub_pd (__m128i a, __m128i b)`

Vor- und Nachteile

- ▶ Abstraktion von Assembly
 - ▶ Bessere Lesbarkeit und Wartbarkeit
 - ▶ Freiraum für Compileroptimierungen
- ▶ Verlust der Plattformunabhängigkeit

Quiz: Vor- und Nachteile

SIMD-Intrinsics beschleunigen meinen Code in jedem Fall.

☐

Ja, ineffizienter SIMD-Code wird weg optimiert

☐

Nein, die neuen Datentypen bringen den Compiler durcheinander.

☐

Nein, bei SIMD-Intrinsics gelten dieselben Abwägungen wie bei SIMD in Assembler.

Automatische Vektorisierung

```
1 void saxpy(long n, float alpha,  
   float* x, float* y){  
2     for(int i = 0; i < n && i <  
       ARR_LENGTH; i++){  
3         y[i] = alpha*x[i] + y[i];  
4     }  
5 }
```

- ▶ GCC automatisch ab -O3
- ▶ -march=native
- ▶ -fopt-info-vec(-missed)
- ▶ objdump

Quiz: number of iterations cannot be computed

Wie kann der Beispielcode trotz Fehler vektorisiert werden?

```
blas_vectorize.c:22:26: missed: not vectorized:  
    number of iterations cannot be computed.
```

☐

Entferne `&& i < ARR_LENGTH`

☐

`i += 4`

☐

Die betroffene Schleife kann nicht vektorisiert werden

Quiz: Optimierung

Kann der Code noch weiter optimiert werden?

☐

Nein

☐

Ja, indem 16 Byte Alignment für x und y erzwungen wird

☐

Ja, indem auch AVX Erweiterungen erlaubt werden

Bonus Quiz: aliasing

Wie kann die folgende Warnung noch entfernt werden?

```
blas_vectorize.c:19:5: optimized: loop versioned  
    for vectorization because of possible aliasing
```

☐

Markiere x und y mit restrict

☐

Bennene y in x um

☐

Die Warnung kann nicht behoben werden