SIDM-Programmierung

Eine Einführung

Robert Clausecker <fuz@fuz.su>
Berliner Linux User Group e. V.

23. März 2025

Einführung

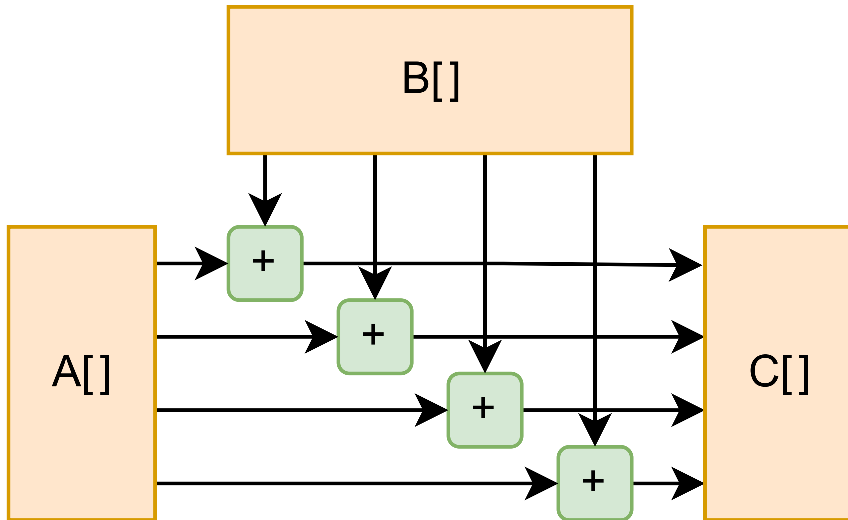
Was ist SIMD?

Einführung

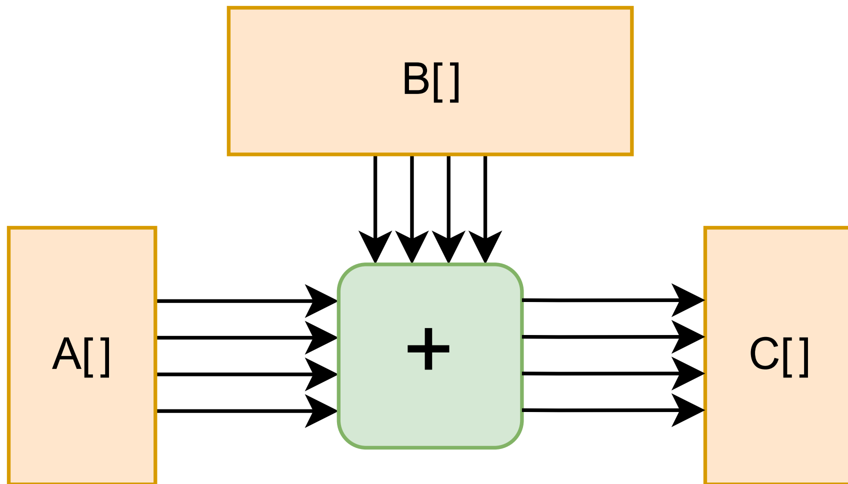
Was ist SIMD?

- **S**ingle **I**nstruction **M**ultiple **D**ata
- Register, die kurze Vektoren von Zahlen halten
- Befehle, die auf diesen Vektoren arbeiten
- mehr Daten pro Befehl $\rightarrow$ höhere Rechenleistung

Einführung



Einführung



Anwendungsgebiete

Die Nutzung von SIMD-Befehlen erhöht die numerische Bandbreite um einen Faktor von 4–16 (je nach Datentyp, Aufgabe, und SIMD-Erweiterung).
SIMD hilft immer da wo das der begrenzende Faktor ist.

- Audio- und Videocodecs
- Numerische Programmierung
- Symmetrische Kryptographie und Hashing
- Bildverarbeitung
- Graphikdarstellung (Software-Renderer)
- … und viele weitere

Anwendungsgebiete

Die Nutzung von SIMD-Befehlen erhöht die numerische Bandbreite um einen Faktor von 4–16 (je nach Datentyp, Aufgabe, und SIMD-Erweiterung).
SIMD hilft immer da wo das der begrenzende Faktor ist.

- Audio- und Videocodecs
- Numerische Programmierung
- Symmetrische Kryptographie und Hashing
- Bildverarbeitung
- Graphikdarstellung (Software-Renderer)
- ... und viele weitere

Voraussetzung: Algorithmus muss Datenparallelität zulassen!
$\rightarrow$ erfordert oft Umdenken im Datenfluss

Datentypen

Mit was für Daten können wir rechnen?

Datentypen

Mit was für Daten können wir rechnen?

Weit verfügbar

- Ganzzahlen: 8, 16, 32 Bit (mit Vorzeichen)
- Gleitkommazahlen: float, double

Manchmal verfügbar, oder nicht mit allen Operationen

- Ganzzahlen ohne Vorzeichen
- 64-Bit-Zahlen
- 16-Bit-Gleitkomma (BF16, FP16)

Tatsächlich verfügbare Typen sind platformabhängig

Verfügbarkeit auf x86

SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, CLMUL

- SSE/SSE2 garantiert verfügbar auf x86-64
- Vektoren zu 128 Bit
- Ganzzahlen bis 64 Bit (mit Vorzeichen), float, double
- ab SSSE3: gute Permutationsbefehle

Verfügbarkeit auf x86

SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, CLMUL

- SSE/SSE2 garantiert verfügbar auf x86-64
- Vektoren zu 128 Bit
- Ganzzahlen bis 64 Bit (mit Vorzeichen), float, double
- ab SSSE3: gute Permutationsbefehle

AVX, AVX2, FMA, F16C

- AVX2 verfügbar seit Haswell (2013)
- alles was SSE kann, dazu vorzeichenlose Zahlen, 16-Bit Gleitkomma (F16C), Vektoren zu 256 Bit, Fused-Multiply-Add (FMA)

Verfügbarkeit auf x86

SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, CLMUL

- SSE/SSE2 garantiert verfügbar auf x86-64
- Vektoren zu 128 Bit
- Ganzzahlen bis 64 Bit (mit Vorzeichen), float, double
- ab SSSE3: gute Permutationsbefehle

AVX, AVX2, FMA, F16C

- AVX2 verfügbar seit Haswell (2013)
- alles was SSE kann, dazu vorzeichenlose Zahlen, 16-Bit Gleitkomma (F16C), Vektoren zu 256 Bit, Fused-Multiply-Add (FMA)

AVX-512 (viele Erweiterungen)

- Vektoren zu 512 bit, viele neue Features
- aber kaum verfügbar

Benutzung im Assembler

```
0:      vmovdqu  (%rdi, %rcx, 4), %ymm0
        vpaddd   (%rsi, %rcx, 4), %ymm0
        vmovdqu  %ymm0, (%rdi, %rcx, 4)
        add      $8, %rcx
        cmp      %rdx, %rcx
        jle      0b
```

Benutzung im Assembler

```
0:      vmovdqu  (%rdi, %rcx, 4), %ymm0
        vpaddd   (%rsi, %rcx, 4), %ymm0
        vmovdqu  %ymm0, (%rdi, %rcx, 4)
        add      $8, %rcx
        cmp      %rdx, %rcx
        jle      0b
```

→ möglich für Experten, aber nicht einsteigerfreundlich

Intrinsische Funktionen

```
#include <immintrin.h>
__m256i _mm256_add_epi32(__m256i a, __m256i b);
```

Intrinsische Funktionen

```
#include <immintrin.h>
__m256i _mm256_add_epi32(__m256i a, __m256i b);
```

`__m256i` Vektortyp: 256-Bit-Vektor von Ganzzahlen

`_mm256` Intrisische Funktion auf 256-Bit-Vektoren

`add` Operation: Addition

`epi32` Datentyp: gepackte 32-Bit-Ganzzahlen mit Vorzeichen

Intrinsische Funktionen

```
#include <immintrin.h>
__m256i _mm256_add_epi32(__m256i a, __m256i b);
```

`__m256i` Vektortyp: 256-Bit-Vektor von Ganzzahlen
`_mm256` Intrisische Funktion auf 256-Bit-Vektoren
`add` Operation: Addition
`epi32` Datentyp: gepackte 32-Bit-Ganzzahlen mit Vorzeichen

$\rightarrow$ `_mm256_add_epi32()` addiert zwei Vektoren zu 8 Ganzzahlen à 32 Bit

Daten bewegen

Basisoperationen zur Bewegung und Erzeugung von Vektoren

- Erzeugung von Vektoren: `setr` (jedes Element verschieden), `set1` (jedes Element gleich)
- Vektoren laden: `loadu` (load unaligned)
- Vektoren schreiben: `storeu` (store unaligned)
- Einzelne Elemente einfügen / extrahieren: `insert`, `extract`
- Elemente aus verschiedenen Quellen laden: `gather`

Beispiel:

```
__m256i iota = _mm256_setr_epi32(0, 1, 2, 3, 4, 5, 6, 7);
__m256  pi = _mm256_set1_ps(3.1415926f);
double roots[4] = { 1, 1.414213562373095O, 2, 1.7320508075688772 };
__m256d vroots = _mm256_loadu_pd(roots);
```

Vertikale Operationen

Vertikale Operationen verarbeiten jedes Element eines Vektors mit dem entsprechenden Element eines anderen.

$$(a_0, a_1, a_2, a_3) + (b_0, b_1, b_2, b_3) = (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$$

- Arithmetik: `abs`, `add`, `avg`, `min`, `max`, `sub`, `mul(lo)`, `div` (nur für FP)
- Logik: `and`, `andnot`, `or`, `xor`, `sll(v)`, `srl(v)`, `sra(v)`
- Vergleiche: `cmp`, `cmpeq`, `cmpgt`, `test`
- Konvertierungen: `cvt…_…`, `ceil`, `floor`
- Funktionen: `sqrt`, `rcp` (reziproker Wert), `rsqrt`
- … und viele mehr

Vertikale Operationen (Beispiel)

```
/* mit -mavx2 kompilieren */
void vector_sum(int32_t *restrict a, const int32_t *b, size_t n)
{
    size_t i;

    for (i = 0; i + 8 <= n; i += 8) {
        __m256i a0 = _mm256_loadu_si256((__m256i *)(a + i));
        __m256i b0 = _mm256_loadu_si256((__m256i *)(b + i));
        a0 = _mm256_add_epi32(a0, b0);
        _mm256_storeu_si256((__m256i *)(a + i), a0);
    }

    for (; i < n; i++)
        a[i] += b[i];
}
```

Permutationen

Permutationen bewegen Elemente umher, ohne sie zu ändern.

- Kombination von zwei Vektoren nach Maske: `blend(v)`
- Ein Element in alle Elemente kopieren: `broadcast`
- Permutationen innerhalb eines Vektors: `unpack`, `permute`, `shuffle`, `bslli`, `permutevar`
- Permutation zwischen zwei Vektoren: `permute2x128`, `palignr`
- Achtung: Verfügbare Permutationen stark abhängig von der Elementgröße

Permutationen (Beispiel)

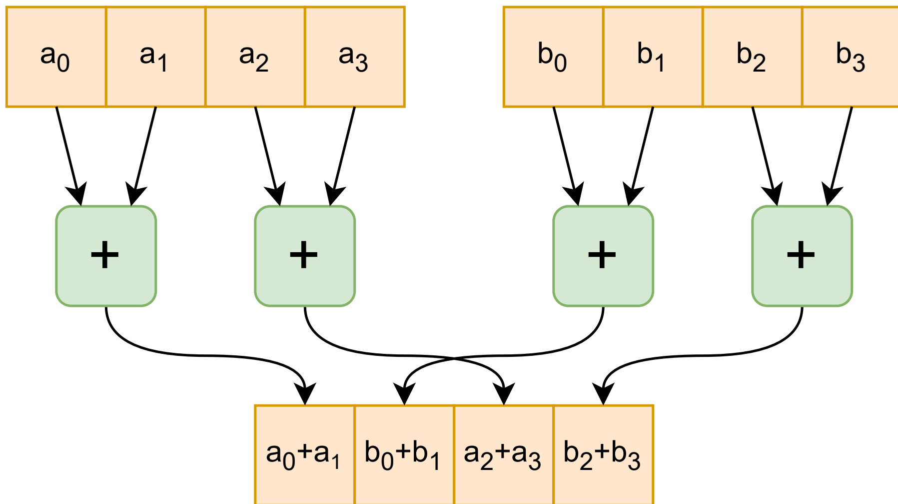
```
/* (r0, i0, r1, i1, ..., r3, i3) -> (r0^2 + i0^2, ..., r3^2 + i3^2) */
__m128 complex_norm(__m256 v)
{
    /* v = (r0, r1, r2, r3, i0, i1, i2, i3) */
    __m256i perm = _mm256_setr_epi32(0, 2, 4, 6, 1, 3, 5, 7);
    v = _mm256_permutevar8x32_ps(v, perm);
    __m128 real = _mm256_extractf128_ps(v, 0); /* (r0, r1, r2, r3) */
    __m128 imag = _mm256_extractf128_ps(v, 1); /* (i0, i1, i2, i3) */
    real = _mm_mul_ps(real, real);
    imag = _mm_mul_ps(imag, imag);

    return (_mm_add_ps(real, imag));
}
```

Horizontale Operationen

Horizontale Operationen verarbeiten die Elemente eines Vektors miteinander.

- paarweise Operationen: adjazente Elemente werden miteinander verrechnet (`hadd`, `hsub`, `madd`)
- Reduktionen: alle Elemente werden miteinander verrechnet (nicht bei x86)
- Kompression: Elemente auswählen die man behalten will (ab AVX-512)
- Expansion: Elemente werden an angegebene Stellen verteilt (ab AVX-512)
- verschiedene Spezialbefehle …

Horizontale Operationen (_mm256_hadd_pd)



Horizontale Operationen (Beispiel)

```
/* (a0, ..., a3) * (b0, ..., b3) -> a0*b0 + a1*b1 + ... + a3*b3 */
double inner_product(__m256d a, __m256d b)
{
    a = _mm256_mul_pd(a, b); /* a0b0, a1b1, a2b2, a3b3 */
    __m128d lo = _mm256_extractf128_pd(a, 0); /* a0b0, a1b1 */
    __m128d hi = _mm256_extractf128_pd(a, 1); /* a2b2, a3b3 */
    lo = _mm_hadd_pd(lo, hi); /* a0b0 + a1b1, a2b2 + a3b3 */

    /* a0b0 + a1b1 + a2b2 + a3b3 */
    return (_mm_cvtsd_f64(_mm_hadd_pd(lo, lo)));
}
```

Unterschiede zwischen den Architekturen

SIMD ist auf jeder Architektur ein bisschen anders.

- Intrics sind komplett architekturspezfisisch
- $\rightarrow$ SIMD-Code muss für jede neue Architektur angepasst werden

Unterschiede zwischen den Architekturen

SIMD ist auf jeder Architektur ein bisschen anders.

- Intrics sind komplett architekturspezfisisch
- $\rightarrow$ SIMD-Code muss für jede neue Architektur angepasst werden
- ARM/ARM64: NEON / ASIMD / SVE
  - Vektoren zu 128 Bit / variabler Länge
  - Unterschiede in vertikalen Operationen
  - Umfangreiche Speicherzugriffsmöglichkeiten
  - andere Rundungen und Konvertierungen
- PowerPC: Altivex / VMX / VSX
  - Vektoren zu 128 Bit
  - Speicherzugriffe müssen ausgerichtet sein
  - Big-Endian-Organisation (seltsam)
  - umfangreiche horizontale Operationen
- RISC-V: RVV (variable Länge)

Unterschiede zwischen den Architekturen (ARM-Beispiel)

```
/* (a0, a1) * (b0, b1) -> a0*b0 + a1*b1 */
double inner_product64(float64x2_t a, float64x2_t b)
{
    a = vmulq_f64(a, b); /* a0*b0, a1*b1 */

    return (vaddvq_f64(a)); /* a0*b0 + a1*b1 */
}

/* (a0, ..., a3) * (b0, ..., b3) -> a0*b0 + a1*b1 + ... + a3*b3 */
float inner_product32(float32x4_t a, float32x4_t b)
{
    a = vmulq_f32(a, b); /* a0*b0, a1*b1, a2*b2, a3*b3 */

    return (vaddvq_f32(a)); /* a0*b0 + a1*b1 + a2*b2 + a3*b3 */
}
```

Portable Abstraktionen

Was, wenn wir uns diesen Aufwand nicht machen wollen?

Portable Abstraktionen

Was, wenn wir uns diesen Aufwand nicht machen wollen?

Frameworks für SIMD in portabel:

- Google Highway
  - High-Level API für datenparalleles programmieren
  - Guter Code auf vielen Platformen
- SIMD everywhere
  - implementiert intrinsische Funktionen der Platformen auf anderen Platformen
  - SSE-Code läuft auf ARM usw.
- Web Assembly SIMD

Aber: Performanzverlust! Die Profis (FFmpeg, OpenSSL, ...) nutzen den Assembler

Autovektorisierung

Was, wenn wir uns diesen Aufwand nicht machen wollen?

Autovektorisierung

Was, wenn wir uns diesen Aufwand nicht machen wollen?

Der Compiler kann automatisch SIMD nutzen.

```
void vector_sum(int32_t *restrict a, const int32_t *b, size_t n)
{
    for (i = 0; i < n; i++)
        a[i] += b[i];
}
```

Funktioniert oft, ist aber unzuverlässig.
$\rightarrow$ Generierten Assemblercode inspizieren!

Weiterführendes Material

Intel Intrinsics Guide
https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Agner Fog's Optimization Manuals
https://agner.org/optimize/

Intel Software Development Manuals