

# Das SIMD-Konzept

- ▶ Bisher: Verarbeiten von Daten im Speicher der Reihe nach
- ▶ Wiederholte Ausführung der selben Instruktionen
- ▶ **Leitfrage:** Gibt es eine Möglichkeit Daten abseits von Threading parallel zu verarbeiten?

# SISD: Single Instruction Stream Single Data Stream

- ▶ Instruktion arbeitet auf Daten bestimmter Länge (*byte, word, dword, qword*)
- ▶ Verarbeiten eines Array: Iterieren über Speicherbereich
- ▶ **Beispiel:** Addition zwei Vektoren

# Beispiel: Vektoraddition

e0   e1   e2   e3   e4   e5   e6   e7   e8   e9   e10   e11   e12

0	1	2	3	4	5	6	7	8	9	10	11	12
+	+	+	+	+	+	+	+	+	+	+	+	+
0	1	2	3	4	5	6	7	8	9	10	11	12
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
0	2	4	6	8	10	12	14	16	18	20	22	24

# Quiz

Wie viele Additions-Operationen benötigt die Addition von zwei Vektoren mit je 2048 32-Bit Integer?

☐

1024 Additionen, da man zwei 32Bit Integer in einem 64Bit Register speichern kann

☐

2048 Additionen, da so viele Integer addiert werden müssen

☐

2047 Additionen und eine Subtraktion

# SIMD: Single Instruction Stream Multiple Data Stream

- ▶ SISD-Iteration funktioniert, aber nicht optimal
  - ▶ Viele Rechenoperationen und hohe Anzahl an Sprungbefehlen
- ▶ Lösung: *Vektorisierung*
  - ▶ Anwendung der selben Instruktion auf ganzen Satz aus Datenobjekten
  - ▶ Neue SSE-Befehle für Vektorverarbeitung
- ▶ Nutzung der bereits aus Float-Berechnung bekannten 128Bit XMM-Register

# Quiz

Wie viele 32-Bit Integer passen in ein XMM-Register?

☐

4

☐

6

☐

8

# Quiz

Wie viele float passen in ein XMM-Register?

☐

6

☐

2

☐

4

# Quiz

Wie viele `uint8_t` passen in ein XMM-Register?

☐

12

☐

16

☐

32



# SSE-Instruktionen für SIMD

- ▶ Neue eigenständige Instruktionen für parallele Datenverarbeitung
- ▶ Instruktionen arbeiten auf ganzem XMM-Register zur gleichen Zeit
- ▶ Auch hier **keine** Speicher zu Speicher Operationen

# SSE-Instruktionen für SIMD

- ▶ Es gibt unterschiedliche Stufen der SSE-Erweiterung
  - ▶ Auf modernen CPUs (x86 und AMD64) immer SSE2 verfügbar
  - ▶ Immer prüfen ob Instruktion auf Plattform verfügbar ist
- ▶ Gesonderte Befehle für Integer und Floating-Point Berechnung
  - ▶ Schlüsselwort: **Packed (P)** - Keine einheitliche Position für Integer und Float (ADDPD/PADDD)
  - ▶ Kein Effekt auf nebenstehende Daten im XMM-Register (Überläufe, etc.)
  - ▶ Ziel in der Regel XMM-Register

# Quiz

Welche Voraussetzungen sind für eine einfache Nutzung von SIMD-Instruktionen bezüglich der Datenbeschaffenheit ideal?

☐

Die Daten liegen nebeneinander und fortlaufend im Speicher

☐

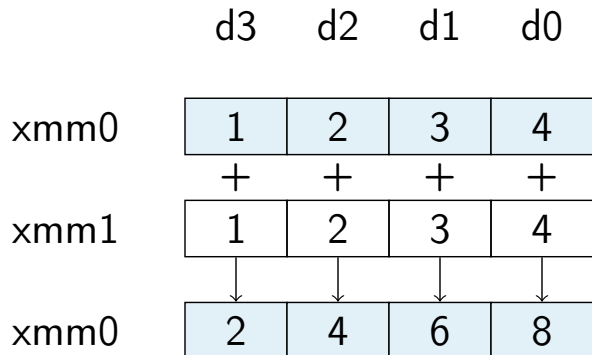
Das XMM-Register kann mit möglichst wenigen Speicherzugriffen gefüllt werden

☐

Nach jedem 16-Byte Block terminiert eine 0 das jeweilige XMM-Glied

# SIMD: Integer Instruktionen

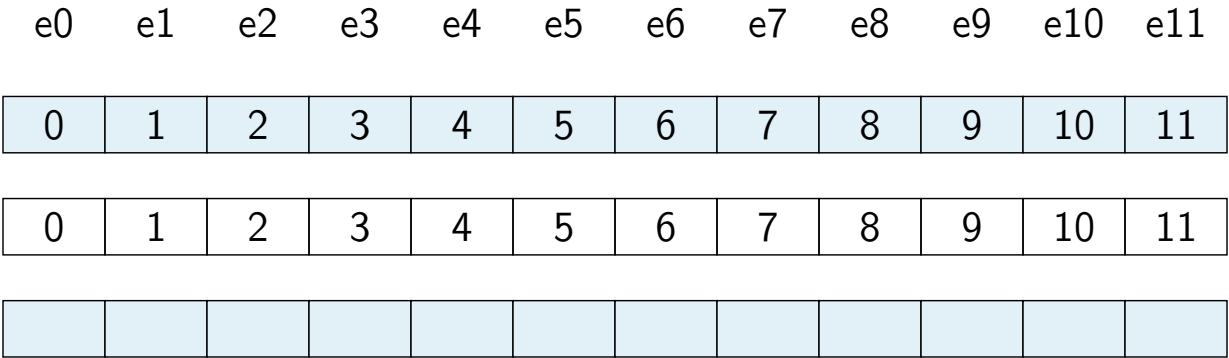
- ▶ `PADDQ xmm1, xmm2/m128` ADD Packed Integer DWORD
  - ▶ Addiert 4 DWORD Integer auf 4 andere DWORD Integer



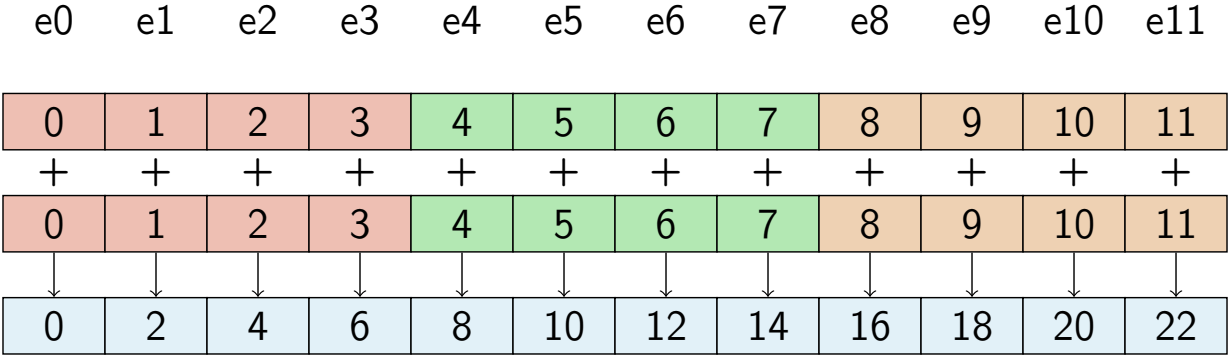
# SIMD: Integer Instruktionen

- ▶ `PADDQ xmm1, xmm2/m128` ADD Packed Integer QWORD
  - ▶ Addiert 2 QWORD Integer auf 2 andere QWORD Integer
- ▶ `PADDB xmm1, xmm2/m128` ADD Packed Integer Byte
  - ▶ Addiert 16 Byte Integer auf 16 andere Byte Integer
- ▶ `PADDW xmm1, xmm2/m128` ADD Packed Integer DWORD
  - ▶ Addiert 4 DWORD Integer auf 4 andere DWORD Integer

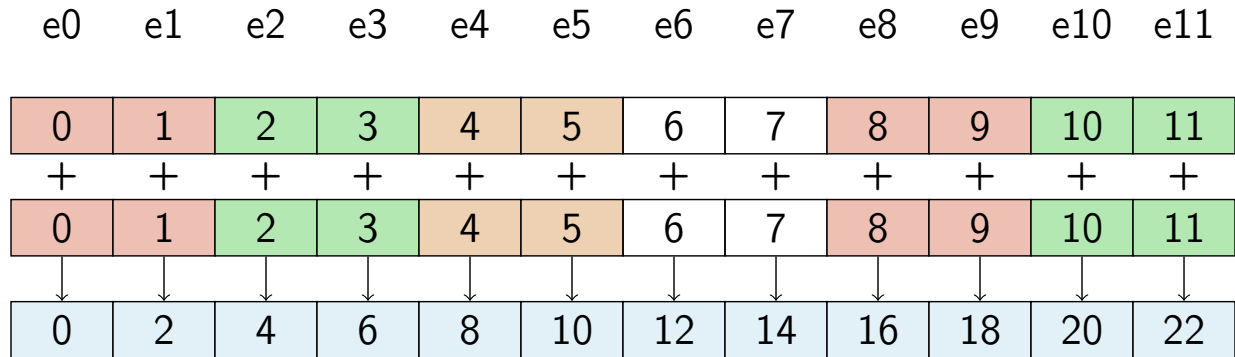
# SIMD-Vektoraddition



# SIMD-Vektoraddition: 32Bit



# SIMD-Vektoraddition: 64Bit





# SIMD: Inkrementieren jedes Elements

```
1 // void add_one(int x[4])
2 add_one:
3     mov     eax, 1
4     movd    xmm0, eax
5     pshufd  xmm0, xmm0, 0x00
6     movdqu  xmm1, [rdi]
7     paddb   xmm1, xmm0
8     movdqu  [rdi], xmm1
9     ret
```

- ▶ MOVD kopiert einen DWORD aus EAX in XMM0
- ▶ PSHUFD kopiert den niedersten DWORD in XMM0 mittels der Shuffle-Maske 0x00 in die drei höherwertigen DWORD in XMM0:  
 $[0, 0, 0, 1] \rightarrow [1, 1, 1, 1]$
- ▶ MOVDQU lädt 4 Integer aus dem Speicher ab RDI in XMM1
- ▶ PADDD addiert 4 Integer aus XMM0 auf XMM1
- ▶ MOVDQU schreibt das Resultat in den Speicher: 4 Integer aus XMM1 in Speicher ab RDI

# Quiz

Angenommen die Vektoren bestünden aus 2048 Byte-Integern.  
Wie oft muss der Befehl PADDB ausgeführt werden?

☐

2048

☐

1024

☐

128

# SIMD: Gleitkomma Instruktionen

- ▶ `ADDSS xmm1, xmm2/m32` ADD **Scalar** Single-Precision Floating-Point Values
  - ▶ Addiert einen Float auf einen anderen Float
  - ▶ Dies ist keine SIMD-Instruktion (**packed**), sondern die bekannte **scalar** Instruktion
- ▶ `ADDPS xmm1, xmm2/m128` ADD Packed Single-Precision Floating-Point
  - ▶ Addiert 4 Float auf 4 andere Float
- ▶ `ADDPD xmm1, xmm2/m128` ADD Packed Double-Precision Floating-Point
  - ▶ Addiert 2 Double auf 2 andere Double

# SIMD-Addition: Vektor auf sich selbst

```
1 // void(float*, size_t)
2 // Annahme: rsi & 3 = 0
3 add_self:
4     movups xmm0, [rdi]
5     addps  xmm0, xmm0
6     movups [rdi], xmm0
7     add    rdi, 16
8     sub    rsi, 4
9     jnz    add_self
10    ret
```

- ▶ MOVUPS lädt 4 Floats ab der Speicheradresse RDI in das Register XMM0
- ▶ ADDPS addiert vier Floats in XMM0 paarweise auf sich selbst
- ▶ MOVUPS schreibt 4 Floats vom Register XMM0 an die Speicheradresse RDI
- ▶ Erhöhen des Zeigers in RDI um 16 Byte und Reduktion des Zählers in RSI um 4

# Alignment

- ▶ Wir haben das SIMD-Konzept kennengelernt und erste Instruktionen hierfür betrachtet
  - ▶ **Offene Frage:** Wie kommen die 128Bit in die XMM-Register?
- ▶ Spezielle neue MOV-Instruktionen für die XMM-Register
- ▶ Allerdings gilt etwas Neues zu berücksichtigen: Das *Data Alignment*

# Data Alignment

- ▶ Programmierer, Compiler und das Betriebssystem besitzen Freiheiten bei der Ablage von Daten im Hauptspeicher
- ▶ Bisher war genaue Position der Daten (Startadresse) im Speicher nicht relevant
- ▶ Bei Nutzung von SIMD-SSE Instruktionen müssen aber aufgrund der Performanz neue *Alignment*-Anforderungen erfüllt werden

# Data Alignment

- ▶ Die meisten SIMD-Instruktionen erfordern ein **16Byte Alignment**
  - ▶ **Skalare** SSE-Instruktionen wie ADDSS tun dies nicht
  - ▶ Adresse muss 16Byte aligned sein, d.h. sie muss teilbar durch 16 sein. In binärer Form sind die niederwertigsten vier Ziffern **0**
  - ▶ Exceptions bei Zugriff auf non-aligned Speicherbereich
  - ▶ Alignment von Variable und Structure-Fields in C mittels:  
`__attribute__((aligned (16)))` z.B:  
`int x[12] __attribute__((aligned (16)));`

# Quiz

Welchen Vorteil bringt die neue Alignment-Anforderung?

☐

Daten können passgenau in eine Cache-Line gebracht und aus dieser abgerufen werden

☐

Der Compiler hat weniger Arbeit bei der Berechnung von Offsets

☐

Konstruktion konzentrischer Caches um ALU nun möglich



# Aligned Zugriff

- ▶ `MOVAPS xmm/m128 xmm/m128` Move Aligned Packed Single-Precision
  - ▶ Kopiert 4 Floats vom Ursprung (Speicher/Register) in das Ziel (Speicher/Register)
  - ▶ Speicher-Operanden müssen *aligned* sein
  - ▶ Kann auch 4 Int, 2 Double oder 2 Long kopieren. Allerdings gibt es dafür gesonderte Instruktionen wie `MOVDQA`
- ▶ **Wichtig:** Speicher-Operanden bei allen SIMD-Instruktionen fordern das *Alignment*

# Unaligned Zugriff

- ▶ `MOVUPS xmm/m128 xmm/m128` Move Unaligned Packed Single-Precision
  - ▶ Kopiert 4 Floats vom Ursprung (Speicher/Register) in das Ziel (Speicher/Register)
  - ▶ Langsamer als `MOVAPS` bei unaligned Zugriff
  - ▶ Ist auf modernen CPUs auf aligned Speicher genauso schnell wie `MOVAPS`
  - ▶ Im Zweifelsfall `MOVUPS` verwenden, bei Kenntnis aber `MOVAPS`

# Quiz

Warum ist ein **unaligned** Zugriff langsamer?

☐

U kommt nach A im Alphabet und somit ist die Instruktion länger in der *Decode-Stage*

☐

Der Befehl wird absichtlich mit einem NOP verlangsamt um Programmierer zu erziehen

☐

Ein 16Byte Block kann sich sonst eventuell über das Ende einer Cache-Line erstrecken

# SIMD Stolperfallen

- ▶ SIMD kann Rechenzeit sparen und Verarbeitung beschleunigen
  - ▶ Auf *günstigen* CPUs aber eventuell langsamer
  - ▶ Höhere Leistungsaufnahme/Temperatur mit möglicher Taktreduzierung
- ▶ SIMD eignet sich nicht für jedes Problem
  - ▶ Zusätzlicher Overhead für SIMD muss sich auch lohnen
  - ▶ SIMD macht einen sub-optimalen Algorithmus nicht automatisch optimal
  - ▶ Variierender *Control-Flow* kann SIMD-Vorteile minimieren

# Compiler und Vektorisierung

- ▶ Compiler versuchen, Programmcode automatisch zu vektorisieren
- ▶ Bis 2020 (nach ca. 20 Jahren) der Entwicklung: Weiterhin schwere Aufgabe für Compiler
  - ▶ Problemstellung ist komplex und das Ergebnis oft nicht optimal

# Compiler und Vektorisierung

- ▶ Wenn Vorteil von SIMD für Problemstellung ersichtlich, dann:
  - ▶ Nicht auf den Compiler verlassen
  - ▶ **Möglichkeit 1:** Assembly wie im Praktikum
  - ▶ **Möglichkeit 2:** Nutzung von Intrinsics
- ▶ **Vorsicht:** Verwendung von Befehlssatzerweiterungen wie SSE, AVX und FMA kann Kompatibilität einschränken