THE SECRETS OF VECTOR API

Plattformunabhängigkeit in einem plattformdiversen Zeitalter

03.2024 - Martin Stypinski, VeeMG GmbH

Vector-Was & Plattform-Dings: 🤯

- Vector API:
  - Java API zur Vektorisierung von Operationen

- Plattform-unabhängigkeit:
  - *Write once, run everywhere*

- Plattform-divers:
  - Intel, AMD, Apple M1, ARM, AWS Graviton, etc.

Dies und das

- Keine Affiliation zu: Intel, Oracle
- Nur API-User, kein Entwickler

- Alle Benchmarks: Java 18 – Incubator III
  - *JDK 18, OpenJDK 64-Bit Server VM, 18+36-2087*
  - Kein offizielles Release Feature, bis jetzt!

- Work in Progress…

Agenda

- Einführung – Kurzer Exkurs, was ist SIMD?
- Vector API – Wie funktioniert das?
- Auto-Vectorization – Weil JVM nicht dumm ist?
- Benchmarks & Beispiele

Flynn's Taxonomy of Computers

	Single Instruction	**Multiple Instruction**
Single Data	Single Instruction, Single Data (SISD) • Uniprocessor (Single-Core)	Multiple Instruction, Single Data (MISD) • FPGA, Google TPU
Multiple Data	Single Instruction, Multiple Data (SIMD) • Vector Computing (GPU) • Vector Extension (MMX, SSE2, etc.)	Multiple Instruction, Multiple Data (MIMD) • Multi-Core, Multi-Processors

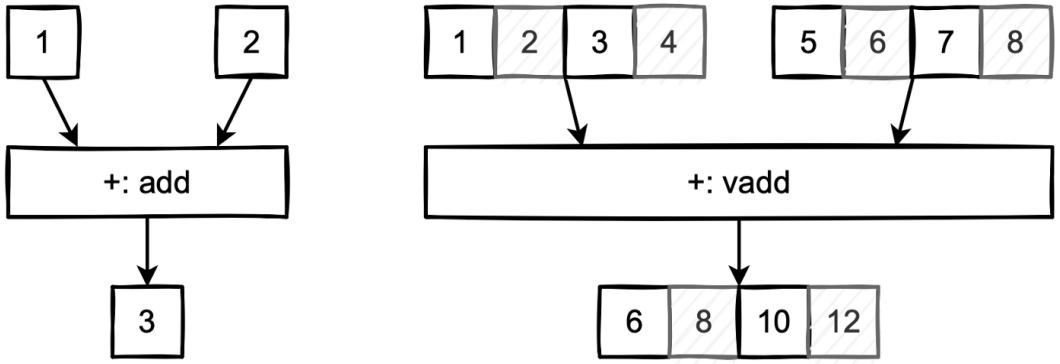
Warum brauchen wir SIMD überhaupt?

- Algorithmen auf Mediendaten:
  - Fixe Operationen – GANZ viel Daten!
- Beispiel Schwarz-Weiss Konvertierung:
  - Grayscale = 0.299R + 0.587G + 0.114B
- Kompression, Verschlüsselung, Audio
- Numerische Methoden

- Vorsicht mit flow-control:
  - If, switch, etc

VeeMG

SIMD Vector Erweiterungen



- Was ist SIMD?
  - Erweiterung der ISA (Instruction Set Architecture)
  - Instruktionen für die *parallele Verarbeitung* (64, 128, 256, 512 bits)
- Warum wird das gemacht?
  - Einfache ‘in-chip’ Implementierung

Performance Benefit - seit 1997

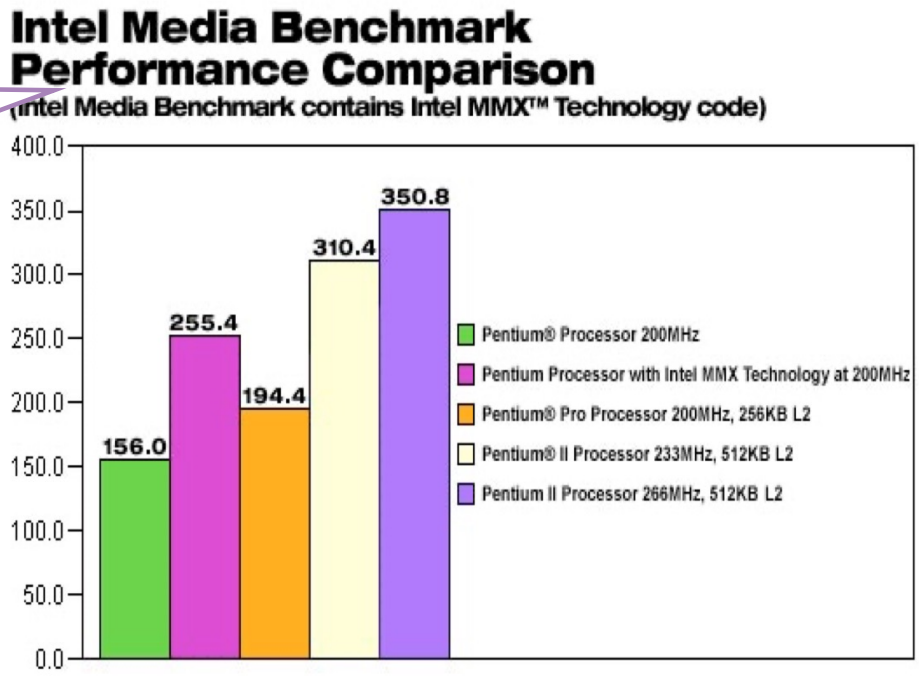
Intel Media Benchmark

Performance Comparison

	Pentium processor 200MHz	Pentium processor 200MHz—MMX Technology	Pentium Pro proc 200MHz 256KB L2		
Overall	**156.00**	**255.43**	**194.38**		
Video	155.52	268.70	158.34	271.98	307.24
Image Processing	159.03	743.92	220.75	1,026.55	1,129.01
3D Geometry*	161.52	166.44	209.24	247.68	281.61
Audio	149.80	318.90	240.82	395.79	446.72

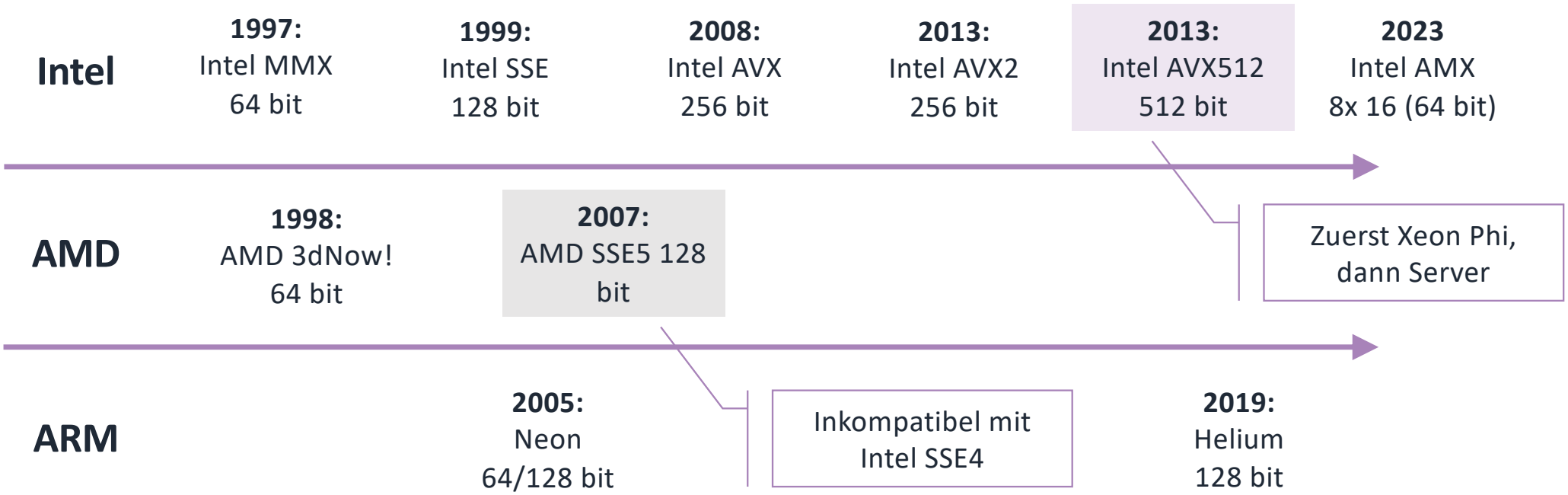
Pentium processor and Pentium processor with MMX technology are measured with 512K L2 cache

4.5x Speed-up!



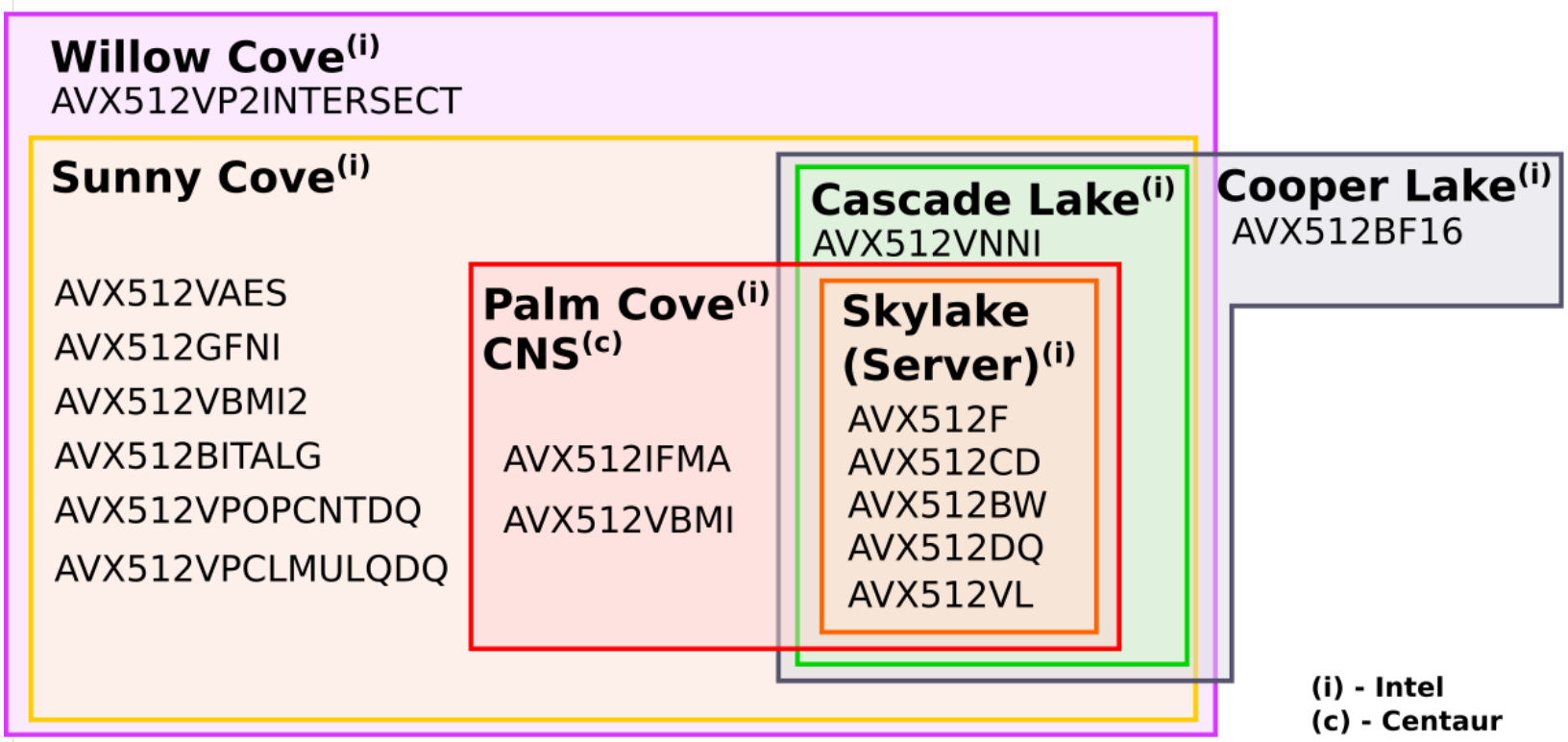
VeeMG

Dieser Jungle: MMX, SSE, AVX, Neon



VeeMG

AVX-512: Was für ein Chaos!



Zwischenfazit

- Viele unterschiedliche Technologien:
  - MMX, SSE, AVX, AVX-512, Neon, AMX (2023)

- Rückwärtskompatibilität:
  - Plattformspezifisch und höchst-optimiert
  - Plattformagnostisch mit klarem Performance up-lift

- «Write once, run everywhere»

Vector-Parallelität: Aber wie?

- Wird benötigt für:
  - Fine grain parallelism
  - Data-intense processing

- Optionen (sortiert nach Aufwand):
  - Bibliotheken (Eg. Numpy)
  - Compiler Vectorization
  - Intrinsics / API
  - Assembler (Hell NO!?! 💣)

Roll your own: Java Vector API / C# Intrinsics

- Java und C# bieten eine Abstraiktionsschicht für SIMD Code:
  - Beide APIs teilen viele Konzepte, Implementationen in C# und Java haben ähnliche Herausforderungen

- C#: Erste Version ca. .NET Core 2.0
  - ARM and AVX2 Verbesserungen in .NET Core 3.0 (2019)

- Java im Moment unter 'Incubator'-Flag
  - **Nicht im offiziellen Release-Deployment** (["--add-modules", "jdk.incubator.vector"])
  - Wird vermutlich *bald* in in den Release aufgenommen *(Java 21?) (03.2022)*
  - *Nope, not yet ☺ (03.2024)*

- **Der Fokus liegt auf Java 21 - (Java 21, Incubator VI, JEP 417)**

Instruktions Typen (Java Vector API)

- Arithmetic Operations:
  - Add(), Sub(), Div(), Mul()
- Bit masking
  - And(), Or(), Not()
- Compare
  - Compare two vectors and return bit-mask with difference
- Casting
  - Cast to short, int, etc.
- Shuffle
  - Shuffling vector (Permute)
  - Important for encryption algorithm (rot13)

Vector API

- Das Vector API hat sich folgende Ziele gesetzt:
  - Clear and concise API
  - Platform agnostic
  - Reliable runtime compilation and performance on x64 and AArch64 architectures – Wo ist ARM?
  - Graceful degradation

- Vector API Programmteile werden in SIMD ausgeführt:
  - Bei fehlender Hardware: Scalar-Code Fallback! - 🎉

- Wir brauchen keine Hardware Kenntnisse
  - Vielleicht für die letzten 10%...
- «Wenn wir kein 10x machen, machen wir’s nicht.»

Array Addition: c[] = a[] + b[]

Annahme:
a.length == b.length

```
public static int[] scalarComputation(int[] a, int[] b) {
    var c = new int[a.length];

    for (var i = 0; i < a.length; i++) {
        c[i] = a[i] + b[i];
    }

    return c;
}
```

Array Addition als Vector API Implementation

```
public static int[] vectorComputation(int[] a, int[] b) {
    var c = new int[a.length];
    int upperBound = SPECIES.loopBound(a.length);

    int i = 0;
    for (; i < upperBound; i += SPECIES.length()) {
        var va = IntVector.fromArray(SPECIES, a, i);
        var vb = IntVector.fromArray(SPECIES, b, i);
        var vc = va.add(vb);
        vc.intoArray(c, i);
    }


    for (; i < a.length; i++) { // Cleanup loop
        c[i] = a[i] + b[i];
    }

    return c;
}
```

Vector API based Array Addition (line 1)

```
private static final VectorSpecies<Integer> SPECIES = IntVector.SPECIES_PREFERRED;
```

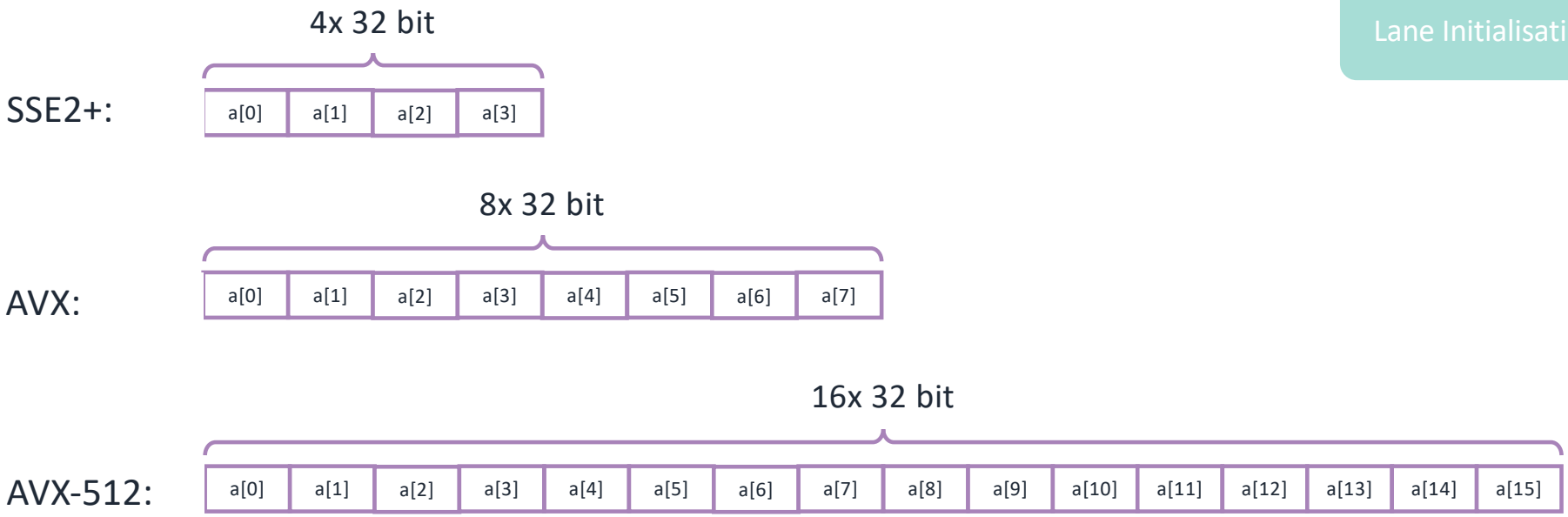
- SPECIES enthält die sogenannte Vektor Länge (aka Vector lane count / Vector Length)
- Diese Typen entsprechen den Java Typen existieren:
  - ByteVector
  - DoubleVector
  - FloatVector
  - IntVector
  - LongVector
  - ShortVector
  - Fast *alle* primitive Datentypen können abgebildet werden (numerisch)
- Verwechselt bitte das Konzept nicht mit C++ Vector<>!

Was ist VectorSpecies<Integer>?

```
private static final VectorSpecies<Integer> SPECIES = IntVector.SPECIES_PREFERRED;
```

Lane Initialisation

- Java Integer Datatype: **32 bit** = **4 byte**

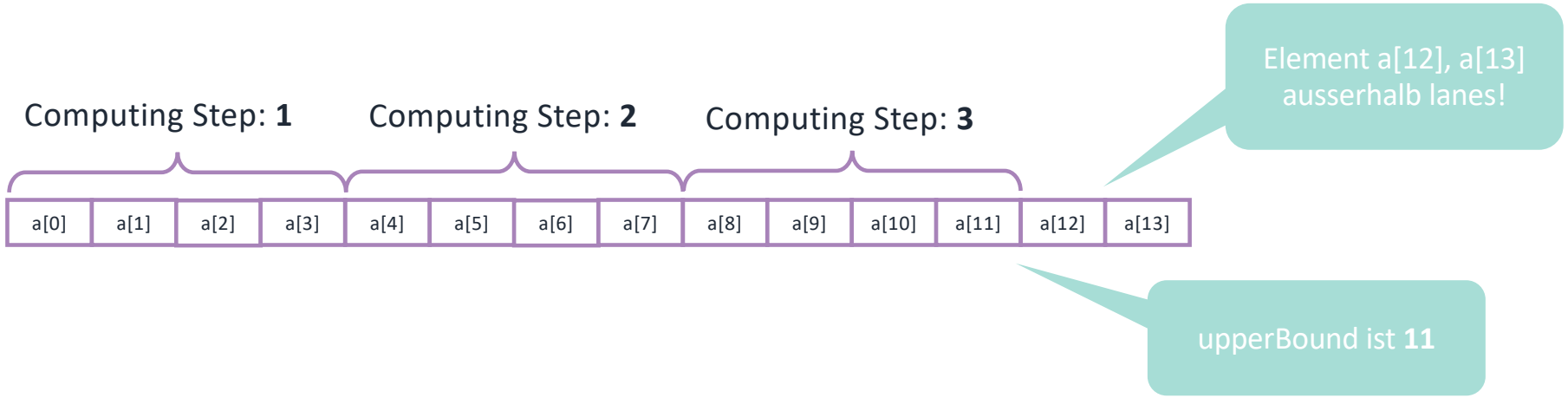


Vector API: Array Addition (Zeile 2-4)

```
public static int[] vectorComputation(int[] a, int[] b) {
    var c = new int[a.length];
    int upperBound = SPECIES.loopBound(a.length);
```

- SPECIES.loopBound – maximale Länge aber passend!

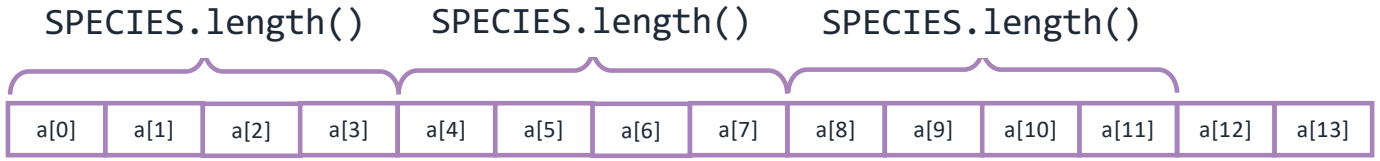
- Unter der Annahme, dass wir SSE (128 bit) verwenden:



Vector API: Array Addition (Zeile 5-10)

```
int i = 0;
for (; i < upperBound; i += SPECIES.length()) {
    var va = IntVector.fromArray(SPECIES, a, i);
    var vb = IntVector.fromArray(SPECIES, b, i);
    var vc = va.add(vb);
    vc.intoArray(c, i);
}
```

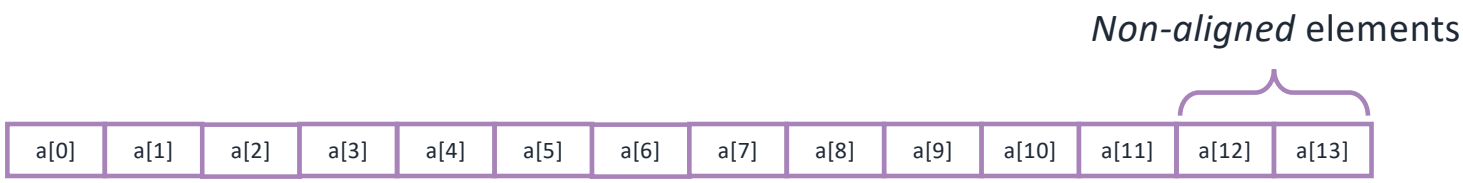
- IntVector.fromArray(SPECIES, array, position) – Erstellen der *Lanes* mit SPECIES.length() Elementen
- Va.add(vb) – Inhalt der Lanes *a* und *b* addieren
- Vc.intoArray() – Zurücklegen der Daten in Array c. – Kein Copy!



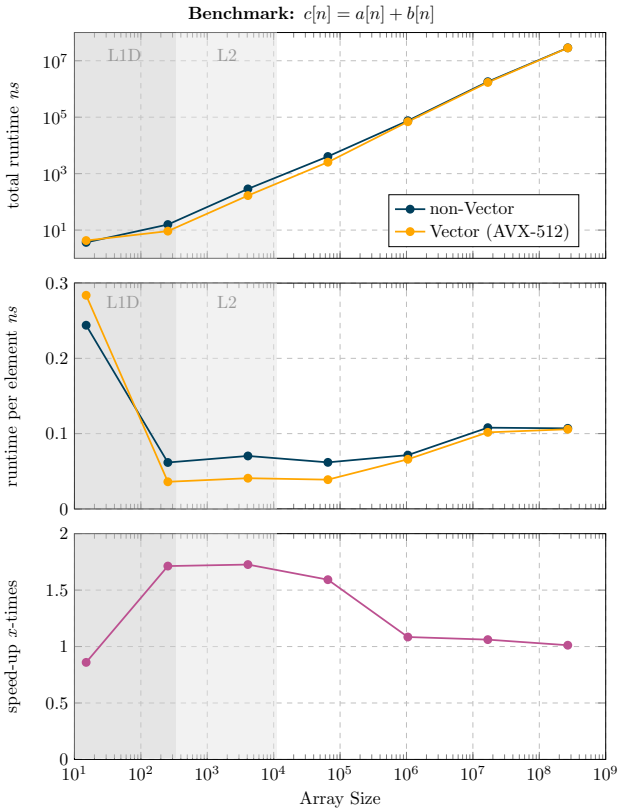
Vector API: Array Addition (Zeile 12-14)

```
for (; i < a.length; i++) { // Cleanup loop
    c[i] = a[i] + b[i];
}
```

- Wir müssen die ‘non-aligned’ Elemente noch abarbeiten!



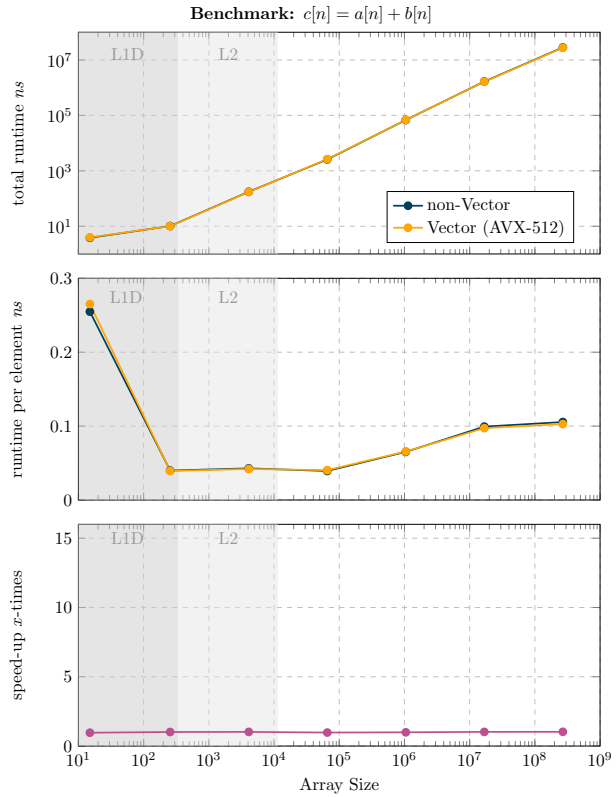
Array-Sum Benchmark



- Peak Speed-up: 1.8x
  - Erwartet: **bis 16x**
- Limitierungen:
  - Latency um AVX-512 zu verwenden
  - Memory vs Compute bound
- Was aus dem L1/2-Cache geht:
  - Performance-Einbruch

Oh da wurde geschumelt... 🤫

Auto-Vectorization

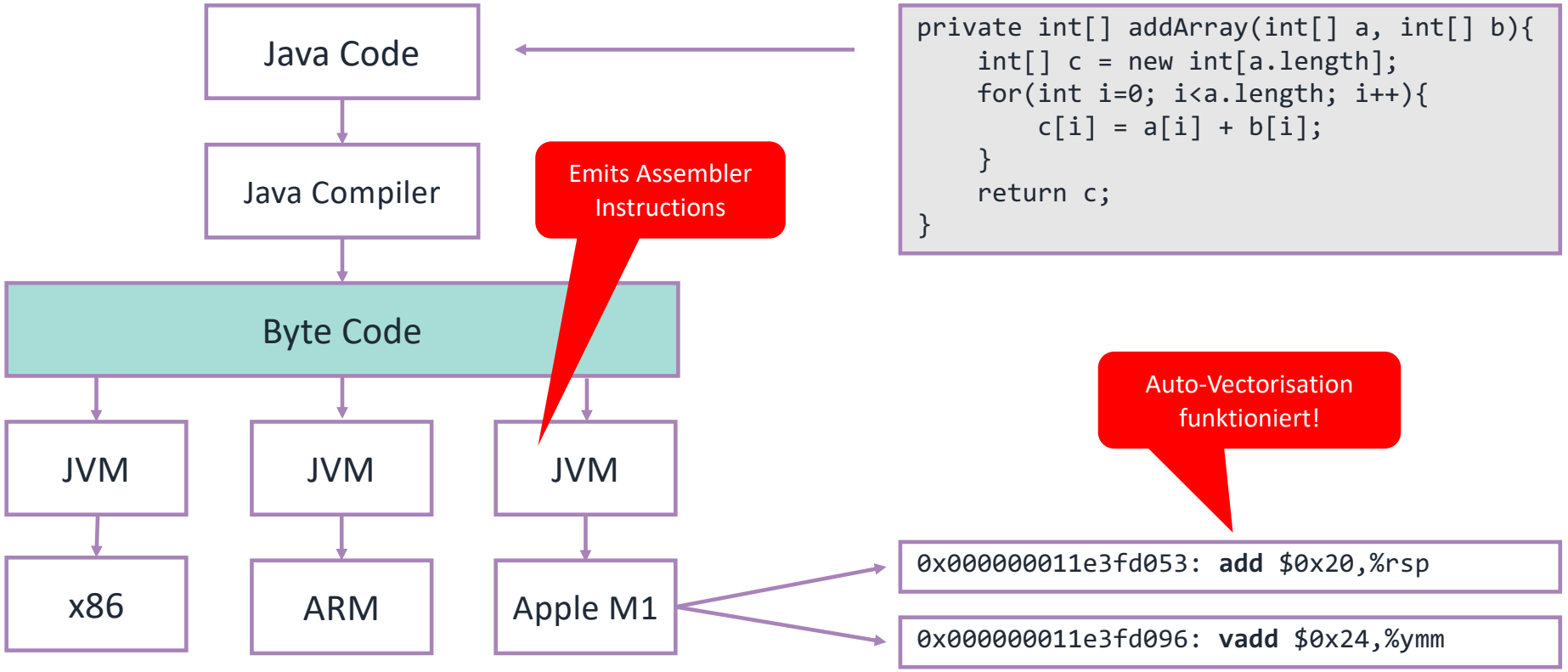


- JVM *macht*: **Auto-Vectorization**:
  - Auto-Vectorization ist das Automatische erzeugen von Vector-beschleunigtem Code. (Bsp. In einfachen for-loops wie hier)
  - Vector API vs JVM: **gleich schnell**

- Das Beispiel wurde erzeugt mit:
  - *-XX:+UseSuperWord / -UseSuperWord*
  - Not guilty of cheating: JVM beats me a lot!

VeeMG

Auto-Vectorization: In-depth



In-depth Tools

- JMH: Java Microbenchmarking Harness
  - Genaues Benchmarken, Reproduzierbar!
- Hsdis.so
  - ByteCode -> ASM
  - Hilft beim Reverse-Engineering des Codes (Vector or Not?)
- PerfASM:
  - Linux Benchmarking Suite
  - Auswertung von Cache Hit, Cache Miss, Branching, etc

Zwischenfazit

- Vector API scheint leserlich, obschon nicht ganz trivial
  - Clean-up Loop, Lanes, Denkweise (i+4)
- Plattformunabhänigkeit ist gewährleistet durch Abstraktion
- Auto-Vectorization ist nicht zu unterschätzen

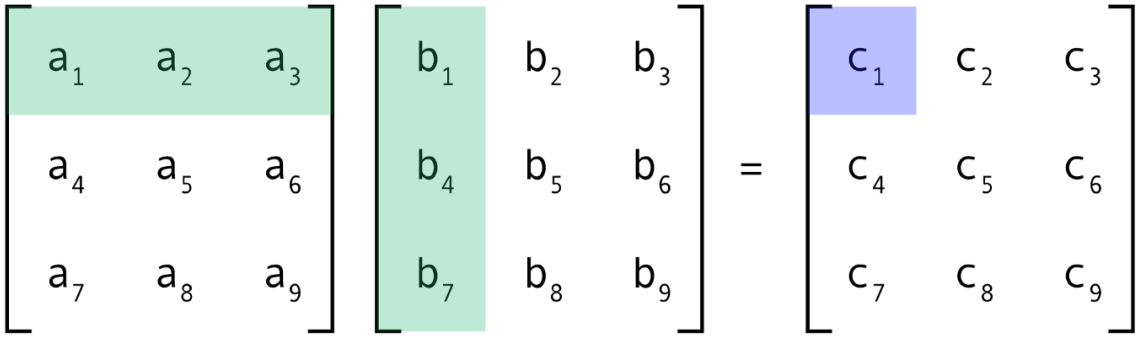
Benchmarks

But Martin…

I think your benchmark is too simple: With one (or two for multiply add) operation, you are just measuring memory bandwith. For "real" code (e.g. a matrix multiplication) you use each element several times and with good blocked multiplication (which.....

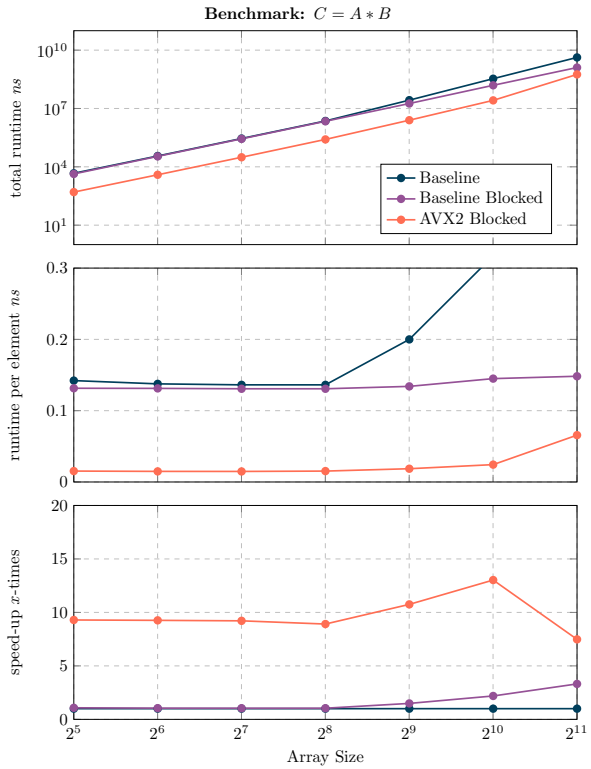
Matrix Berechnung

- A x B; wobei A und B quadratische Matrizen sind:
  - Kantenlänge: [32, ..., 2048]



- c1 = a1 * b1 + a2 * b4 + a3 * b7
- **Laufzeitkomplexität:** O(n^3)

Baseline, Blocked, AVX-2

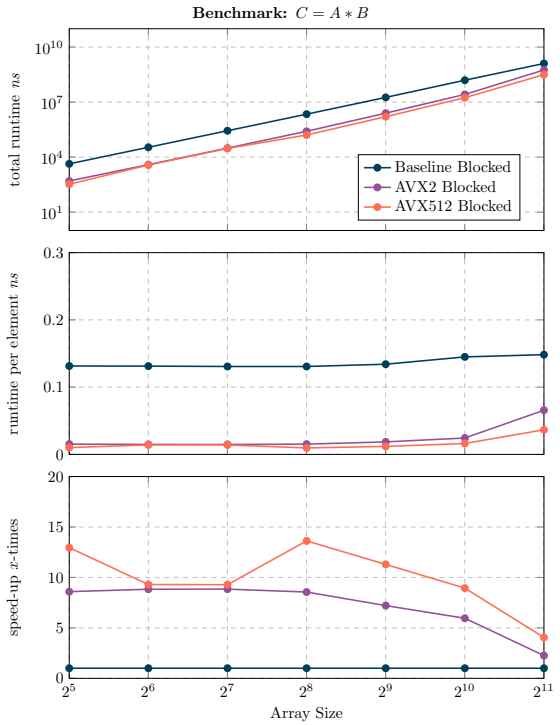


- Baseline:
  Einfachste Implementierung
  3x for-loop
- Blocked:
  Auftrennung in Blöcke
- AVX2 Blocked:
  Blöcke mit Vector API (AVX-2)

- **Blocked** zum eliminieren der L1/L2-Cache Problematik

VeeMG

AVX-2 vs AVX-512: Vector API kanns!

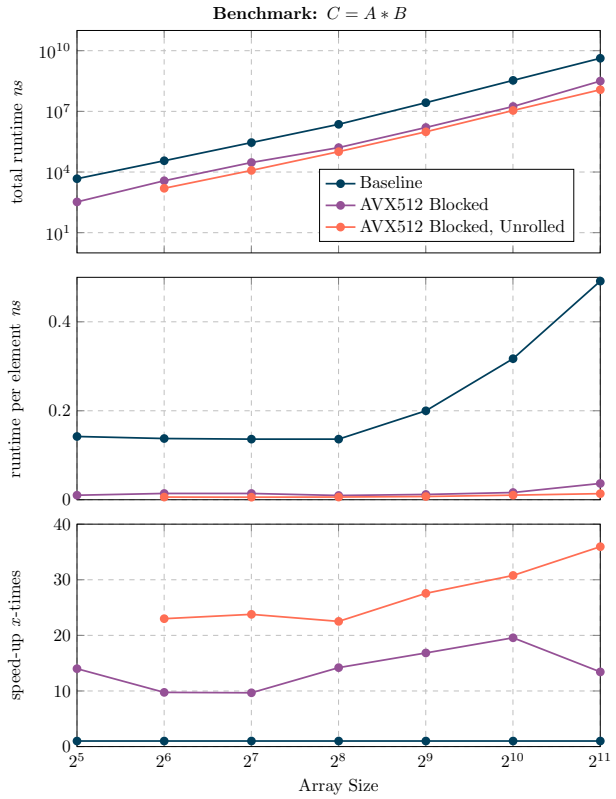


- AVX-2:
  8-10x Speed-Up
- AVX-512:
  10-14x Speed-Up

- Vorsicht: Je nach Plattform verändert sich die Laufzeit!

VeeMG

All in: AVX-512 Unrolled

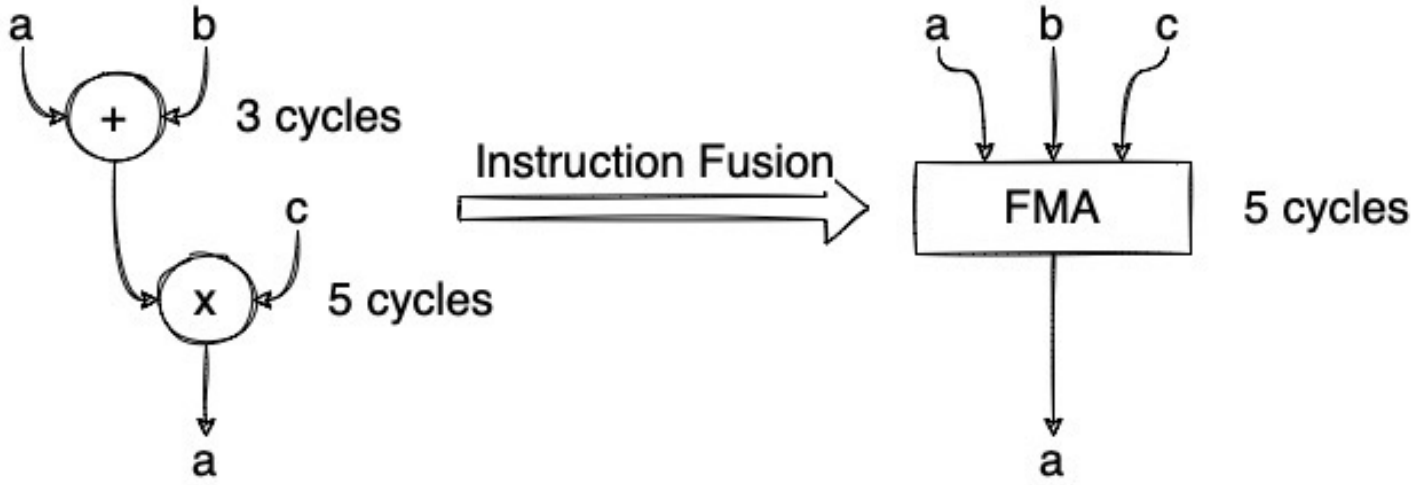


- Baseline
- AVX-512 mit Blocks
  20x Speed-up
- AVX-512 mit Blocks & Unroll
  35x Speed-up

- Höchst optimiert, wird nicht auf allen Plattformen funktionieren!

VeeMG

Fused-multiply add (FMA): a = a + (b x c)



Fazit

- Auto-Vectorisierung funktioniert teilweise so gut, dass man händisch nicht besser wird!

- Vector API macht Sinn für *viele* Fälle
  - «Go for the low hanging fruits»

- Spezifische Implementierung kann *gigantischen* Up-Lift bringen

Fazit II

- SIMD Implementationen sind viel einfacher als GPU Code:
  - Vector API ist sexy! ☺
  - Abstraktion funktioniert sehr gut für verschiedene uArchs
  - Kein Kopieren von Speicher - alles geschieht auf dem Host System

- Your milage may vary

Danke fürs Zuhören & Let’s talk!

- Martin Stypinski

VeeMG