Vektorisierung

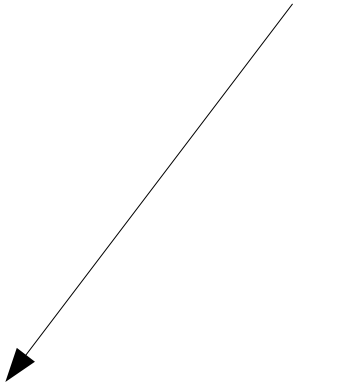
Softwareentwicklung in der Wissenschaft
15.06.2020

Einleitung

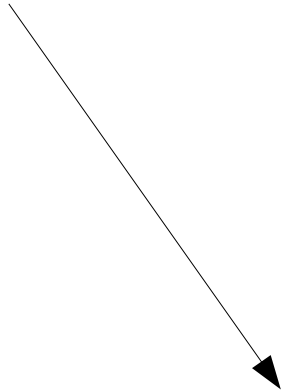
- Motivation
- Definition
- Hardware
- Software
- Programmierbeispiel
- Fazit
- Quellen

Problemstellung

Inhalt eines Arrays vervielfachen



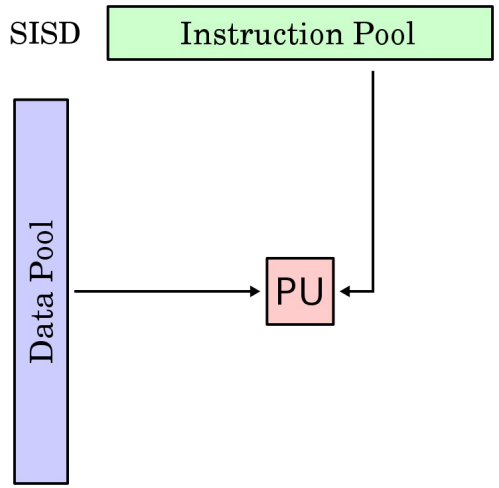
```
1 for(x in 0 to 9)
2 {
3     a[x] = b[x] * 3
4 }
```



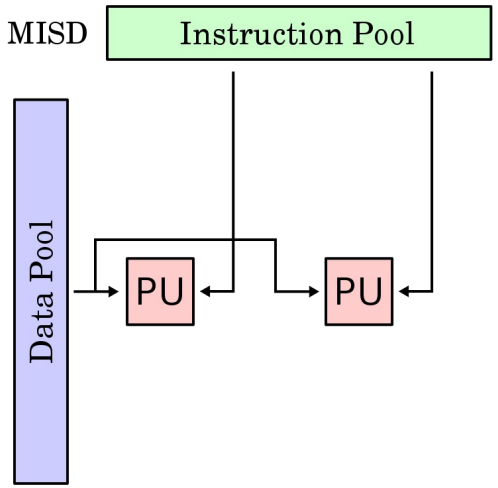
```
1 for(x in 0, 2, 4, 6, 8)
2 {
3     a[x, x+1] = b[x, x+1] * 3
4 }
```

Flynnsche Klassifikation

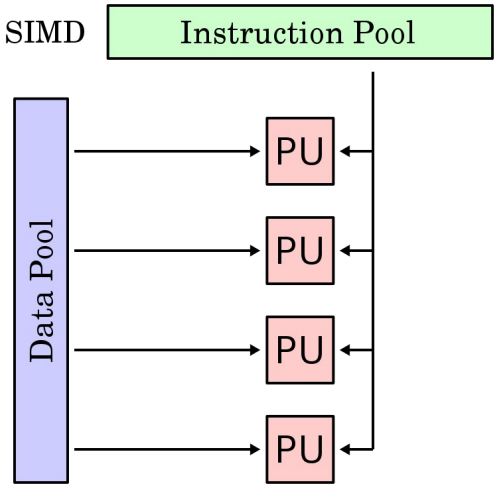
Single **i**nstruction stream,
Single **d**ata stream



Multiple **i**nstruction stream,
Single **d**ata stream



Single **i**nstruction stream,
Multiple **d**ata stream



Multiple **i**nstruction stream,
Multiple **d**ata Stream

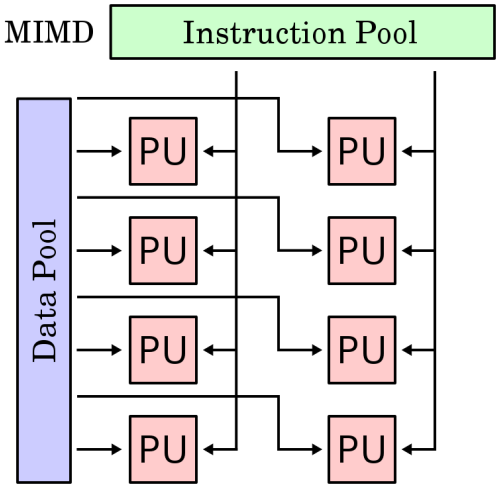


Bild von https://en.wikipedia.org/wiki/Flynn%27s_taxonomy

Allgemeine Informationen

- Vektorisierung: Software so ausführen, dass sie mit mehreren Daten gleichzeitig arbeitet
- Seit ca. 1966 eingesetzt
- Früher in erster Linie für Supercomputer
- Ab 1990 auch in Consumer PCs
- Heute noch weit verbreitet
- Nützlich für große Datenverarbeitung, Vektor- und Matrizenrechnungen

Wieso ist Vektorisierung schneller?

Ohne Vektorisierung:

Führe diese Schleife zehnmal aus:
- Lade die nächste Zahl
- Multipliziere sie mit 3
- Speichere das Ergebnis

Ende der Schleife

Mit Vektorisierung:

Lade die zehn Zahlen

Multipliziere sie mit 3

Speichere das Ergebnis

Vorteile: weniger Befehle geladen, Zahlen schneller geladen, Zahlen schneller verarbeitet

SX Aurora Tsubasa

- Seit 2018 von NEC hergestellt
- Ist kein eigener Supercomputer sonder eine PCIe-Karte
- Vektor Host ist normalerweise ein Linux Server
- Einzeln aber auch in Serverpaketen erwerbbar
- Preis startet bei ca. 6000 Dollar

Aufbau der Karte

- 48 GB HBM2 RAM
- 16 MB LLC
- 8 Kerne mit bis zu 1,6 Ghz/s
- 307 GFLOP pro Kern
- 2,45 TFLOP insgesamt

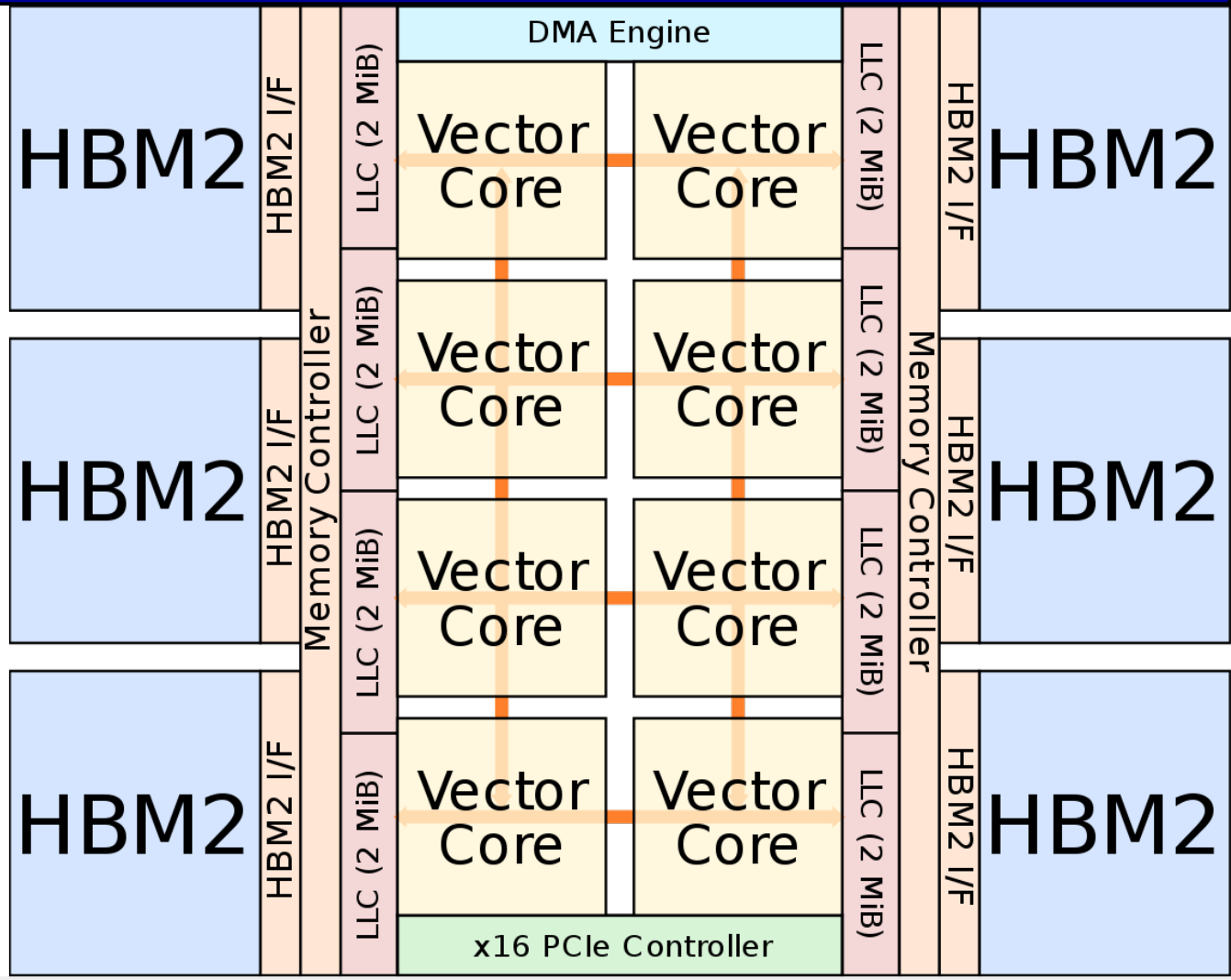
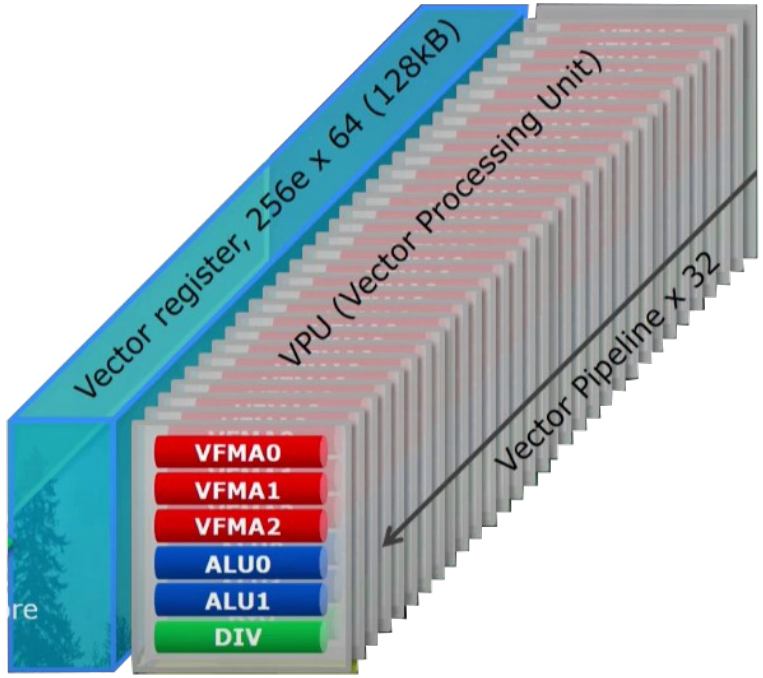


Bild von https://en.wikichip.org/wiki/nec/microarchitectures/sx-aurora

Ein einzelner Kern

- 256 Einträge pro Vektorregister
- Einzelner Beitrag ist 8 Byte groß
- 64 Vektorregister insgesamt
- 32 Berechnungen parallel
- 192 FLOP/Takt



Bilder von https://youtu.be/g6z6nx1h7gI?t=263 und
https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html

Allgemeine Informationen

- Hardware ist essentiell
- Compiler/Interpreter hat das „letzte Wort“
- Es gibt vorgefertigte Methoden, die vektorisieren erleichtern
- Bspw. Bietet Python mit NumPy eine große Bibliothek, mit vektorisierten Funtkionen
- Liefert Funktionen wie sum() oder dot()

Intrinsische Befehle

- Direkter Zugriff auf Befehlssatz des Prozessors
- Aufwändiger als abstrakterer Code
- Prozessor muss Anweisungen unterstützen

```
1  #include "xmmintrin.h"
2
3  void add(float *a, float *b, float *c)
4
5  {
6
7  __m128 t0, t1;
8
9  t0 = _mm_load_ps(a);
10
11 t1 = _mm_load_ps(b);
12
13 t0 = _mm_add_ps(t0, t1);
14
15 _mm_store_ps(c, t0);
16
17 }
```

Bild von https://software.intel.com/content/www/us/en/develop/articles/how-to-vectorize-code-using-intrinsics-on-32-bit-intel-architecture.html

Automatische Vektorisierung

- Code wird vom Compiler automatisch vektorisiert
- Beispiele Intel Compiler für C++, Fortran, GNU Compiler
- Einfache Schleifen werden leicht erkannt
- Nicht jeder Code kann automatisch vektorisiert werden

Richtlinien für Automatische Vektorisierung

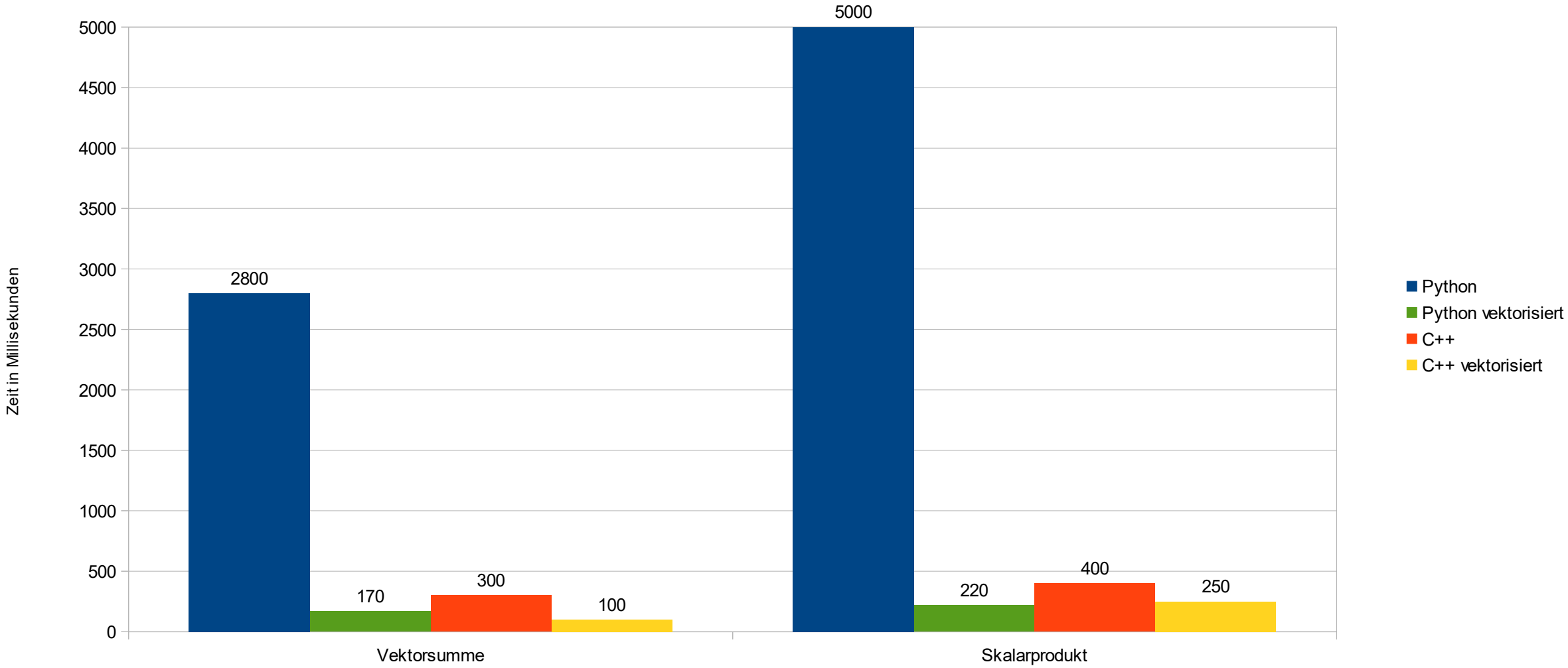
- Wenig Kontrollstrukturen innerhalb von Schleifen
- Einfache for- statt while-Schleifen
- Daten in Arrays speichern
- Datentypen beachten

Code

```
1   int *p = new int[100000000];
2   int *q = new int[100000000];
3   long sum = 0;
4   long dot_product = 0;
5
6   for (int i = 0; i < ; i++)
7   {
8       p[i] = 100000000 + i;
9       q[i] = 200000000 + i;
10  }
11
12
13  #pragma loop(no_vector)
14  for (int i = 0; i < 100000000; i++)
15  {
16      dot_product += p[i] * q[i];
17  }
18
19  #pragma loop(no_vector)
20  for (int i = 0; i < 100000000; i++)
21  {
22      sum += p[i];
23  }
```

```
1   p = numpy.arange(1000000, 2000000)
2   q = numpy.arange(2000000, 3000000)
3
4   dot_product = 0
5   sum = 0
6
7
8
9
10
11
12
13  for i in range(len(p)):
14        dot_product += p[i] * q[i]
15
16  vectorized_dot_product = numpy.dot(p, q)
17
18
19  for i in range(len(p)):
20        sum += p[i]
21
22  vectorized_sum = numpy.sum(p)
23
```

Ergebnisse


Zeit in Millisekunden
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
2800
170
300
100
5000
220
400
250
Vektorsumme
Skalarprodukt
Python
Python vektorisiert
C++
C++ vektorisiert

Problemstellung

Vorteile	Nachteile
- Kann Berechnungen verschnellern...	- aber nicht alle Berechnungen gleichermaßen
- auf dem Großteil aller Rechner benutzbar...	- aber nicht so schnell, wie dafür extra konzipierte Hardware
- Vektorrechner beschleunigen Bearbeitung enorm...	- sind aber teuer anzuschaffen
- Bibliotheken und Funktionen für „alltägliche“ Probleme...	- aber spezifische Probleme können schwerer zu implementieren sein
- automatische Vektorisierung erleichtert die Arbeit...	- aber nicht jeder Code kann automatisch vektorisiert werden

Quellen und Literatur

- https://en.wikipedia.org/wiki/Automatic_vectorization
- https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top.html
- https://en.wikipedia.org/wiki/NEC_SX-Aurora_TSUBASA
- https://en.wikichip.org/wiki/nec/microarchitectures/sx-aurora
- https://sx-aurora.github.io/
- https://www.youtube.com/watch?v=SW60DrfOuGw
- https://fs.hlrs.de/projects/teraflop/28thWorkshop_talks/WSSP28_HKobayashi.pdf
- https://www.tutorialspoint.com/vectorization-in-pythonhttps://www.nec.com/en/global/solutions/hpc/sx/