Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Proseminar Ausarbeitung

# Vektorisierung

vorgelegt von

Benni Möller

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Informatik
Matrikelnummer: 7031401

Betreuer: Jannek Squar

Hamburg, 2020-08-27

# Inhaltsverzeichnis

# 1 Vektorisierung

## 1.1 Motivation

Angenommen ein/e Softwareprogrammierer/in bekommt die Aufgabe gestellt, jeden Eintrag eines Arrays a mit drei zu multiplizieren. Dabei soll das Array die Länge zehn haben. Ein erster Ansatz in Pseudocode könnte so aussehen:

```
1 {
2     for(x in 0 to 9)
3     {
4         a[x] = a[x] * 3
5     }
6 }
```

Listing 1.1: erster Ansatz in Pseudocode

Wenn aber viele, große Datensätze nacheinander oder gleichzeitig bearbeit werden, ist diese Lösung des Problems eventuell zu langsam. Betrachtet man den Code genauer, sieht man nun, das an jedem Element des Arrays die gleiche Operation ausgeführt wird. Gäbe es die Möglichkeit, zwei Einträge gleichzeitig zu bearbeiten, könnte die Laufzeit des Programms effektiv halbiert werden. Umgesetzt könnte diese Idee etwa so aussehen:

```
1 {
2     for(x in 0, 2, 4, 6, 8)
3     {
4         a[x, x+1] = a[x, x+1] * 3
5     }
6 }
```

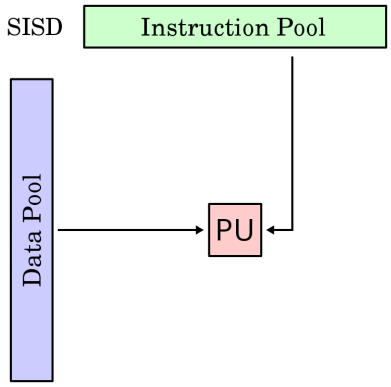
Listing 1.2: zweiter Ansatz in Pseudocode

Hiermit ist auch schon der Grundgedanke der Vektorisierung gegeben. Dieser befasst sich mit Operationen, die an mehreren einzelnen Elementen nacheinander ausgeführt werden. Genauer sollen diese Anweisungen so umgeformt werden, dass sie an mehreren Daten gleichzeitig durchgeführt werden.

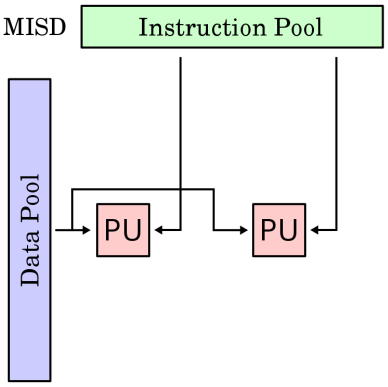
## 1.2 Flynnsche Klassifikation

Bevor Vektorisierung formal definiert wird , ist es hilfreich, zuerst einige Begrifflichkeiten zu klären. Als erstes wird die Flynnsche Klassifikation erläutert. [1]

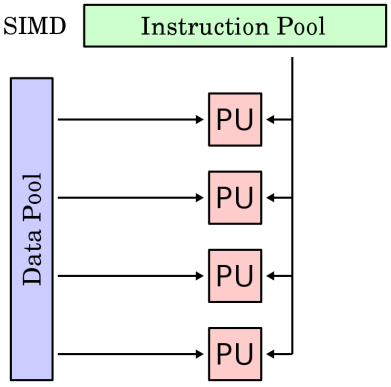
Diese vier Bilder zeigen verschiedene Techniken, wie Prozessoren Informationen bearbeiten können. Der Instruction Pool sind alle Anweisungen, die ein Prozessor ausführen kann, wie beispielsweise Zahlen addieren oder multiplizieren. Die PUs (Processing Units) sind hingegen die Prozessorkerne, die die Anweisungen an den Daten (Data Pool) ausführen kann. Nun ist es, wie in der Motivation bereits angeschnitten wurde, möglich mehrere Daten gleichzeitig zu verarbeiten. Dabei werden bei SIMD und MIMD eine Aktion, die die CPU ausführt, an mehreren Daten ausgeführt. Weiterhin können auch wie bei MISD und MIMD mehrere Informationen gleichzeitig ausgeführt werden. Für das Thema der Vektorisierung ist aber insgesamt SIMD am interessantesten.



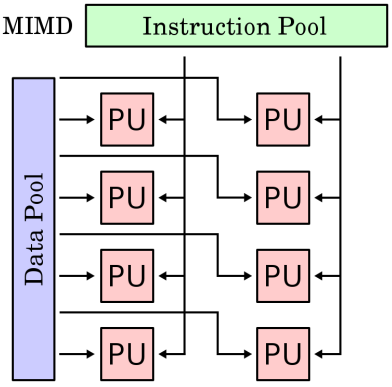

Single instruction stream, single data stream




Multiple instruction streams, single data stream




Single instruction stream, multiple data streams




Multiple instruction streams, multiple data streams

Abbildung 1.1: Wikipedia, Flynnsche Klassifikation

## 1.3 Allgemeine Informationen zur Vektorisierung

Nun kann man mit Hilfe der Flynnschen Klassifikation auch Vektorisierung akkurat definiert werden. Hierbei werden die SIMD und MIMD Befehle eines Prozessors benutzt, um mehrere Daten gleichzeitig zu bearbeiten. In erster Linie tut man dies, um die Geschwindigkeit von Software und deren Berechnungen zu verbessern.

Vektorisierung gibt es seit ca. 1966 und wurde entwickelt, um im den Supercomputer ILLIAC IV benutzt zu werden. [2] Damals waren SIMD-Anweisungen etwas, das nur in Supercomputern Benutzung fand. In den 1990er Jahren wurden aber erstmals auch Consumer PCs leistungsfähig genug, um bspw. Videospiele oder Audiobearbeitung in Echtzeit auszuführen. Damit konnten dann auch SIMD-Anweisungen das erste mal für die breite Masse an normalen Kunden attraktiv werden. [3] Auch heute werden von den meisten Prozessorherstellern noch SIMD-Anweisungen in ihren CPUs verbaut. Wenn man sich die Advanced Vector Extensions (AVX) ansieht, sind diese und deren Nachfolger einem Großteil der AMD und Intel Chips seit 2011 enthalten. [4] [5]

Letztendlich stellt sich noch die Frage, wieso Vektorisierung schneller ist. Um diese Frage anschaulich zu beantworten, bietet sich das Beispiel 1.1 aus der Motivation an.

Als erstes wird dabei das Beispiel betrachtet, dass komplett aus SISD-Anweisungen besteht. Dabei wird die Schleife insgesamt zehn mal ausgeführt. In jedem dieser Durchläufe muss eine Zahl aus dem Array geladen werden. Außerdem muss diese Zahl noch mit drei multipliziert und danach wieder gespeichert werden.

Dem gegenüber steht die Variante des Programms, das SIMD-Anweisungen benutzen kann. Angenommen, dass das Programm von einem Prozessor ausgeführt wird, der zehn Zahlen gleichzeitig multiplizieren kann. Dann würde es gar keine Schleife mehr geben, sondern es würden einfach alle Zahlen gleichzeitig geladen und am Ende auch gleichzeitig gespeichert werden. Dies alleine sorgt schon dafür, dass das Programm schneller ausgeführt werden kann. Der größte Geschwindigkeitsgewinn ist aber dadurch gegeben, dass alle Zahlen gleichzeitig multipliziert werden.

Insgesamt ist Vektorisierung also zeitsparend, wenn die gleichen Berechnungen auf mehreren Daten gleichzeitig ausführt werden. Hierbei können dann mehrere Berechnungen in der gleichen Zeit ausgeführt werden wie eine einzelne Berechnungen.

# 2 SX-Aurora TSUBASA

## 2.1 Allgemeine Informationen

Der SX-Aurora TSUBASA ist ein Vektorprozessor, der von der Firma NEC seit dem Jahr 2017 hergestellt wird. [6] Insgesamt stellt NEC schon seit den 1980er-Jahern Supercomputer her. [7] Besonders an dieser Erfindung ist jedoch, dass sie kein eigener funktionierender Computer ist. Vielmehr ist sie eine PCIe-Karte, die auf ein Mainboard gesteckt werden kann. [8] Als solches funktioniert sie nur in Kombination mit einem Server, auf dem in der Regel Linux installiert ist. [9] Solch ein Server wird Vector Host genannt und kann insgesamt bis zu acht dieser Karten halten. [10] Diese Konstruktion wurde gewählt, um auch einzelnen Unternehmen Vektorprozessoren anbieten zu können, die sich keine Supercomputer leisten können. Eine SX-Aurora TSUBASA Karte kann für circa 6000 US-Dollar direkt von NEC bezogen werden. [11] Wahlweise bietet NEC auch größere Serverkombinationen an, in die diese Karten bereits eingebaut sind.

Wichtig für das Funktionieren des Vektorprozessors ist, dass die Berechnungen möglichst getrennt auf der Karte selbst aufgeführt werden. Je mehr Daten zwischen der SX-Aurora TSUBASA transferiert werden müssen, während das Programm läuft, desto langsamer werden die Berechnungen. Damit dies möglichst einfach umzusetzen ist, gibt es eigene Kompilierer für C++ und Fortran, die von NEC zur Verfügung gestellt werden. [12]

Unter anderem wurde diese Technologie auch in Deutschland verwendet. Der Deutsche Wetterbund hat einen Vertrag mit NEC unterzeichnet und will deren SX-Aurora TSUBASA-Karten benutzen, um bis zum Ende von 2020 deren momentane Rechenleistung zu verdreifachen. [13]

## 2.2 Aufbau der Hardware

In der Abbildung 2.1 sieht man den Aufbau einer SX-Aurora TSUBASA Karte. Eine dieser Karten enthält 6 High Bandwidth Memory 2 (HBM2) Blöcke. Jeder dieser Blöcke enthält 8 GB RAM. Insgesamt sind also auf einer SX-Aurora TSUBASA Karte 48 GB Speicher enthalten. Besonders wichtig für die Technologie ist, dass dieser Speicher schnell angesprochen werden kann. Insgesamt werden so Übertragungsraten von bis zu 1,35 TB/s ermöglicht. Weiterhin sieht man die 8 bis 10 CPU-Kerne. Diese arbeiten mit einer Taktfrequenz von jeweils 1,6 GHz und schaffen es, 307 GF Berechnungen mit doppelter Präzision bzw. 614 GF Berechnungen mit einfacher Präzision zu bewerkstelligen. Außerdem kann jeder einzelne Kern noch auf einen Last Level Cache (LLC) von 2 MiB zugreifen, sodass auf einer Karte insgesamt 16 Mib LLC-Speicher verbaut ist. Letztendlich

sieht man unten auf der Karte noch den 16x PCIe-Anschluss, der in das Mainboard gesteckt wird. [14]

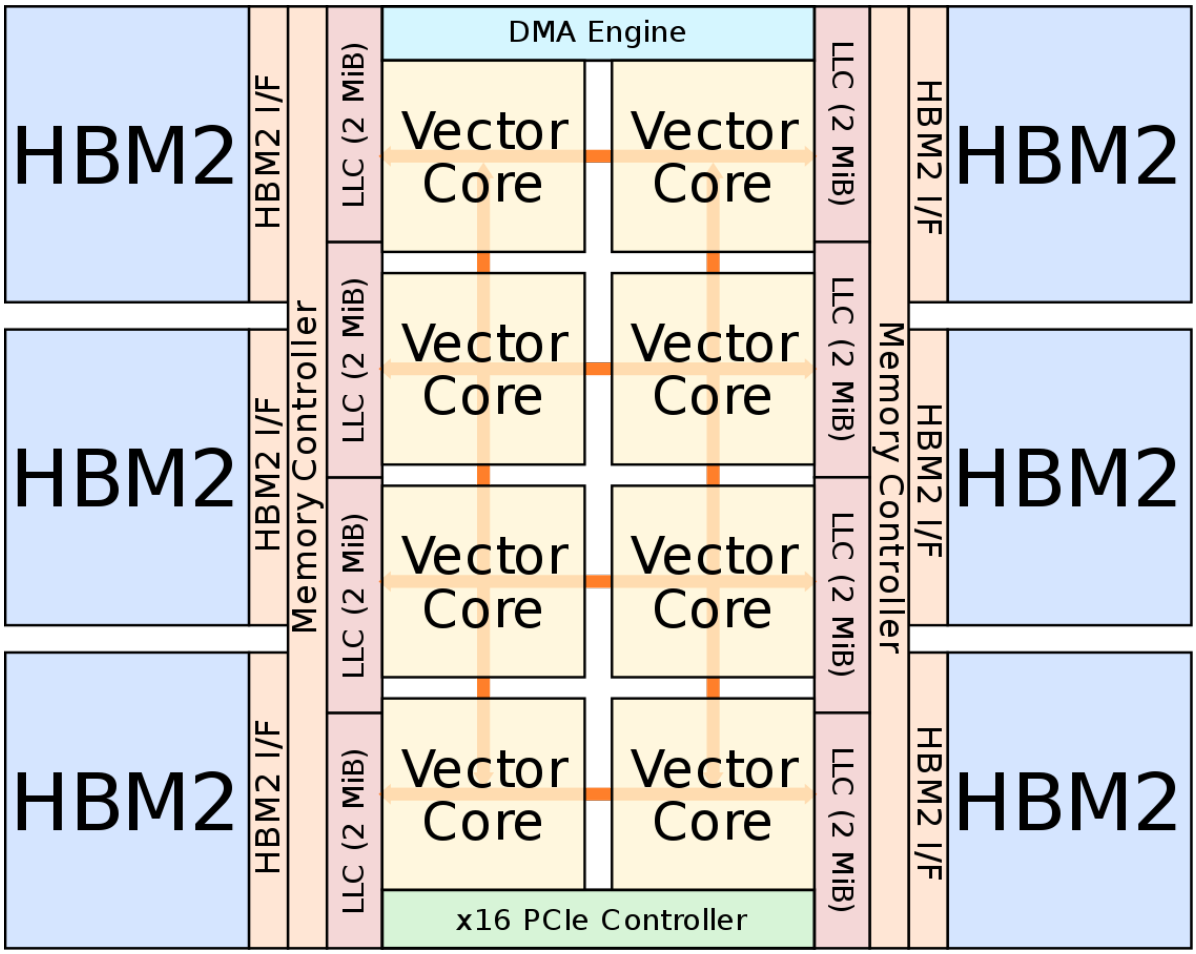



Abbildung 2.1: Wikichip, SX-Aurora TSUBASA, Aufbau der Karte

## 2.3 Aufbau eines einzelnen Kerns

Im folgenden Abschnitt wird der Aufbau eines einzelnen Kerns genauer beschrieben. Die Struktur dieser Komponente ist zu einem großen Teil dafür verantwortlich, weshalb die SX-Aurora TSUBASA-Technologie SIMD-Anweisungen schnell und effizient ausführen kann.

Ein wichtiger Teil eines einzelnen Kerns sind die Vektorregister. Eines dieser Register besitzt 256 Einträge, wobei jeder einzelne Eintrag 8 Byte groß ist. Da jeder Kern 64 Vektorregister enthält, kann so ein einzelner Kern bis zu 16.384 Doubles gleichzeitig speichern. [15] Direkt daran angebunden sind die Vektor Pipelines und VPUs (Virtual Processing Unit). Hier werden in erster Linie alle logischen Operationen ausgeführt. Man hat drei VFMA (Vector Float Multiplay Addition)Abschnitte. Hiermit können

Rechnungen wie A * x + B schnell ausgeführt werden. Mit den beiden ALU Einheiten können andere Operationen wie bspw. das Addieren von Zahlen ausgeführt werden. [15]

Hiermit wird auch klar, weshalb die SX-Aurora TSUBASA Technologie vektorisierte Befehle schnell ausführen kann. Mit drei FMA-Einheiten können aufgrund der 64 Vektorregister pro Takt 192 FLOPs ausgeführt werden. [15] Betrachtet man im Gegensatz dazu eine CPU, die mit den aktuellen AVX-512 Instruktionen ausgestattet ist, kann diese nur 4 FLOPs pro Takt in doppelter Genauigkeit berechnen. [16] Insgesamt kann die SX-Aurora TSUBASA also aufgrund ihrer drei VFMA-Einheiten und ihrer großen Vektorregister deutlich mehr Zahlen verarbeiten.

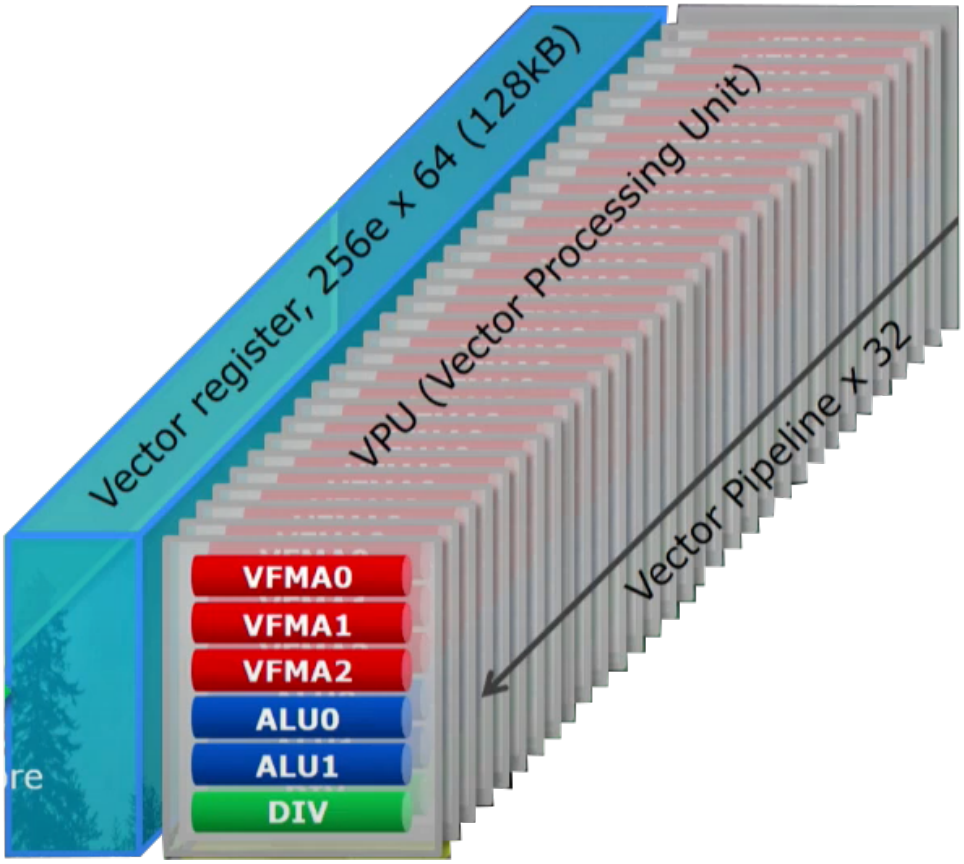



Abbildung 2.2: YouTube, SX-Aurora TSUBASA, Vortrag vom HPC User Forum

# 3 Tipps zu Vektorisierung

## 3.1 Allgemeine Informationen

Die richtige Hardware ist essentiell, um vektorisierte Software auszuführen. Ohne die nötigen SIMD-Instruktionen ist es nicht möglich, Vektorisierung zu benutzen. Andererseits ist auch der Compiler wichtig. Wenn man Code in einer Sprache schreibt, wird sie vom Compiler in Maschinenbefehle übersetzt. In welche Befehle genau ist aber von dem spezifischem Compiler abhängig. Unterstützt es der Compiler dann nicht, SIMD-Instruktionen zu benutzen, kann auch kein vektorisierter Code ausführt werden. Nichtsdestotrotz gibt es einige Möglichkeiten in verschiedenen Sprachen und Compilern, die das Schreiben von vektorisiertem Code erleichtern.

Als erstes gibt es einige Packages oder Bibliotheken für verschiedene Sprachen, die Funktionen liefern, die bereits vektorisiert sind. Python besitzt mit NumPy beispielsweise ein Package, das es ermöglicht auf mehrdimensionalen Arrays zu arbeiten. [17] So werden mit den Funktionen multiply(a, b) zwei Arrays multipliziert oder mit dot(a, b) kann das Kreuzprodukt gebildet werden.

Außerdem bieten einige Compiler die Möglichkeit, die SIMD-Anweisungen der CPU direkt anzusprechen. [18] In Listing 3.1 sieht man, wie mit dem C++ Compiler von Intel mehrere Variablen des Typs Float addiert werden. Hierbei werden in Zeile vier die zwei Speicherplätze t0 und t1 für Floats im Vektorregister freigegeben. In den Zeilen fünf und sechs werden dann die beiden Floats a und b in die freigegebenen Stellen geladen. Letztendlich werden in Zeile sieben die Floats addiert und abschließend in Zeile acht das Ergebnis gespeichert.

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

Listing 3.1: intel.com, How to Vectorize Code Using Intrinsics on 32-Bit Intel® Architecture

## 3.2 Automatische Vektorisierung

Problematisch kann es sein, wenn man Berechnungen ausführen will, für die es keine vorgefertigten vektorisierten Methoden gibt und das benutzen von intrinsischen Funktionen zu lange dauert. Hierbei gibt es aber eine angenehme Lösung.

Automatische Vektorisierung ist eine Technik, bei der ein Compiler nicht vektorisierten Code in vektorisierten Maschinencode umwandelt. Beispiele sind hierbei der Intel Compiler [19] für C++, der GNU Compiler [20] aber auch NEC bietet so einen Compiler für die SX-Aurora TSUBASA Technologie an. [12]

Während bei der automatischen Vektorisierung simple Schleifen gut erkannt werden können, kann es aber schwieriger für den Compiler sein, aufwendigeren Code korrekt als vektorisierbar zu erkennen. Hierfür gibt es aber einige Richtlinien, an die man sich halten kann. [21]

Als Erstes sollten unnötige Kontrollstrukturen innerhalb von Schleifen vermieden werden. Das heißt möglichst wenige if-else, return oder go-to Abzweigungen benutzen. Gleichzeitig sollte man auch Funktionsaufrufe eher sparsam verwenden. Außerdem ist es vorteilhaft, Datensätze in Arrays zu speichern, um dem Compiler die Möglichkeit zu geben, direkt auf die einzelnen Elemente zuzugreifen. Schlussendlich ist es noch wichtig, Datentypen mit Bedacht zu wählen. Verwendet man beispielsweise Doubles, obwohl nur Floats nötig wären, kann das die Effizienz des Programms verringern. Da Doubles doppelt so viel Speicherplatz belegen, können nur halb so viele Zahlen in die Vektorregister geladen werden, wie es theoretisch möglich ist.

## 3.3 Programmierbeispiel

Um zu demonstrieren, wie einfach automatische Vektorisierung ist und wie viel Vektorisierung im Allgemeinem hilft, befindet sich in diesem Abschnitt ein simples Beispiel.

Hierbei wird der Code in Visual Studio 2019 geschrieben , welches den Intel Compiler für C++ benutzt. Dabei sollen alle Einträge eines Vektors aufsummiert und das Skalarpodukt von zwei anderen Vektoren berechnet werden. Die zu testende Variable soll dabei die Ausführungsgeschwindigkeit des Programms werden.

Alles was nötig war, um automatische Vektorisierung zu verwenden, war in den Projekteinstellungen unter dem Reiter C/C++ -> Optimierung die Optimierungseigenschaft auf O2 zu stellen. Wahlweise kann man sich von der IDE anzeigen lassen, welche Schleifen vektorisiert werden und weshalb oder weshalb nicht diese vektorisiert wurden. Außerdem ist es möglich mit dem Befehl #pragma loop(no_vector) die Schleifen von der automatischen Vektorisierung auszuschließen.

Betrachtet man die Ergebnisse des Tests, sieht man schnell, dass die automatische Vektorisierung deutlich schneller gewesen ist. Während das Berechnen des Skalarprodukts ca. 1,5 mal so schnell war, hat das Aufrechnen des Skalarprodukts sogar nur ein Drittel der Zeit gebraucht. Letztendlich ist noch zu erwähnen, dass der Code auf einem Intel

m3-7y30 Prozessor ausgeführt wurde. Da dieser nur AVX-2 Instruktionen besitzt, kann er nur 4 FLOPs mit doppelter Genauigkeit pro Takt berechnen. Mit spezieller Hardware wie einer SX-Aurora TSUBASA Karte ist es vermutlich möglich, deutlich zeitsparendere Vektorisierung zu betreiben.

```
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 < 100000000; i++)
7  {
8      p[i] = 100000000 + i;
9      q[i] = 200000000 + i;
10 }
11
12
13 for (int i = 0; i < 100000000; i++)
14 {
15     sum += p[i];
16 }
```

Listing 3.2: Code für das Programmierbespiel in C++ ausgeführt auf einem Intel m3-7y30

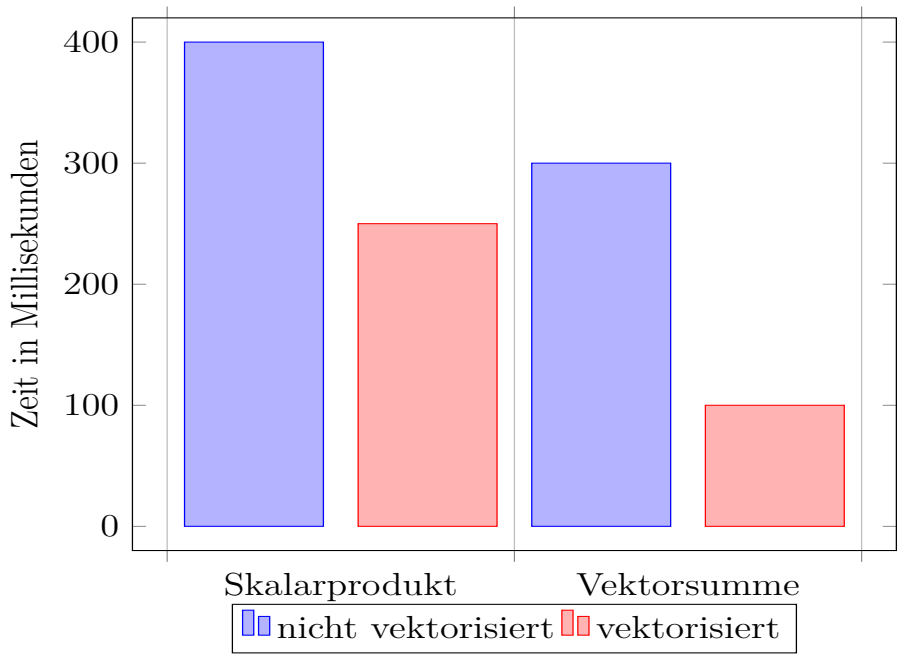



Abbildung 3.1: Ergebnisse des Programmierbeispiels

# 4 Zusammenfassung

Durch Vektorisierung ist es möglich, gewisse Berechnungen deutlich zu beschleunigen. Andererseits können nicht alle Arten von Kalkulationen in gleichem Maße profitieren. Wie bereits erklärt basiert Vektorisierung in erster Linie auf SIMD-Anweisungen und funktioniert am besten, wenn ein Befehl auf vielen Daten gleichzeitig angewendet wird. Damit man diese also verwenden kann, müssen die Daten richtig vorliegen und der Prozessor muss diese SIMD-Anweisungen unterstützen.

Weiterhin gibt es spezielle Hardware, die einzig darauf ausgelegt ist, vektorisierte Programme schnell zu berechnen. Diese Hardware kann ist aber unter Umständen sehr kostenintensiv. So kostet eine SX-Aurora TSUBASA Karte ca. 6000 US-Dollar, während nach oben hin mit riesigen Supercomputern und Serverfarmen praktisch keine Grenzen gesetzt sind.

Auch kann es schwierig sein, Vektorisierung softwaretechnisch umzusetzen. Allgemeinere Probleme wie Berechnungen auf Vektoren oder Matrizen sind oft in Bibliotheken oder Erweiterungen bereits umgesetzt. Bei besonderen Problemen kann auch automatische Vektorisierung helfen, die Ausführung von Code zu beschleunigen. Hierbei ist aber auch eine gewisse Expertise nötig, um es dem Compiler zu ermöglichen, den Code zu erkennen.

Insgesamt ist Vektorisierung eine Technik, die sehr lohnenswert erscheint. Wenn man die nötige Hardware und das erforderliche Wissen besitzt, kann man Berechnungen effizient erledigen. Aber auch ohne spezielle Vektorprozessoren und weniger Expertise lässt sich ein deutlicher Geschwindigkeitssprung durch Vektorisierung erzeugen.

# Literatur


[1] M. J. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* C-21.9 (1972), S. 948–960.

[2] R.M. Hord. *The Illiac IV: The First Supercomputer*. Springer Berlin Heidelberg, 2013, S. 5–8. ISBN: 9783662103456. URL: https://books.google.ca/books?id=mG2rCAAAQBAJ.

[3] G. Conte, S. Tommesani und F. Zanichelli. “The long and winding road to high-performance image processing with MMX/SSE”. In: *Proceedings Fifth IEEE International Workshop on Computer Architectures for Machine Perception*. 2000, S. 302–310.

[4] David Kanter. *Intel's Sandy Bridge Microarchitecture*. Sep. 2010. URL: https://www.realworldtech.com/sandy-bridge/6/ (besucht am 25. 08. 2020).

[5] Joel Hruska. *Analyzing Bulldozer: Why AMD's chip is so disappointing - Page 4 of 5 - ExtremeTech*. Okt. 2011. URL: https://www.extremetech.com/computing/100583-analyzing-bulldozers-scaling-single-thread-performance/4 (besucht am 25. 08. 2020).

[6] *NEC releases new high-end HPC product line, SX-Aurora TSUBASA: Press Releases | NEC*. Okt. 2017. URL: https://www.nec.com/en/press/201710/global_20171025_01.html (besucht am 23. 08. 2020).

[7] *SX-1, SX-2-Computer Museum*. URL: http://museum.ipsj.or.jp/en/computer/super/0008.html (besucht am 23. 08. 2020).

[8] *SX-Aurora TSUBASA Architecture*. URL: https://www.nec.com/en/global/solutions/hpc/sx/architecture.html? (besucht am 26. 08. 2020).

[9] *NEC SX-Aurora TSUBASA Software*. URL: https://www.nec.com/en/global/solutions/hpc/sx/software.html (besucht am 26. 08. 2020).

[10] Timothy Prickett Morgan. *Can Vector Supercomputing Be Revived?* Okt. 2017. URL: https://www.nextplatform.com/2017/10/26/can-vector-supercomputing-revived/ (besucht am 23. 08. 2020).

[11] Andreas Stiller. *NEC SX Aurora TSUBASA: Vektorrechner mit 24 oder 48 GByte High Bandwidth Memory | heise online*. Nov. 2017. URL: https://www.heise.de/newsticker/meldung/NEC-SX-Aurora-TSUBASA-Vektorrechner-mit-24-oder-48-GByte-High-Bandwidth-Memory-3892046.html (besucht am 27. 08. 2020).

[12] *Programming Languages, Tools*. URL: https://www.nec.com/en/global/solutions/hpc/sx/tools.html? (besucht am 27. 08. 2020).

[13] *NEC Receives EUR50 Million Order from Deutscher Wetterdienst (DWD)*. Juni 2019. URL: https://www.finanznachrichten.de/nachrichten-2019-06/46967330-nec-receives-eur50-million-order-from-deutscher-wetterdienst-dwd-011.htm (besucht am 27. 08. 2020).

[14] *NEC SX-Aurora TSUBASA - Vector Engine*. URL: https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html? (besucht am 27. 08. 2020).

[15] *SX-Aurora TSUBASA Architecture*. URL: https://www.nec.com/en/global/solutions/hpc/sx/architecture.html? (besucht am 23. 08. 2020).

[16] Joe Staples. *What Is Intel AVX-512 and Why Does It Matter? - Prowess Consulting*. Jan. 2018. URL: https://www.prowesscorp.com/what-is-intel-avx-512-and-why-does-it-matter/ (besucht am 27. 08. 2020).

[17] *What is NumPy? — NumPy v1.19 Manual*. URL: https://numpy.org/doc/stable/user/whatisnumpy.html (besucht am 27. 08. 2020).

[18] *How to Vectorize Code Using Intrinsics on 32-Bit Intel® Architecture*. Sep. 2011. URL: https://software.intel.com/content/www/us/en/develop/articles/how-to-vectorize-code-using-intrinsics-on-32-bit-intel-architecture.html (besucht am 21. 06. 2020).

[19] *Intel Compiler Vectorization | High Performance Computing*. URL: https://hpc.llnl.gov/software/development-environment-software/intel-compiler-vectorization (besucht am 26. 08. 2020).

[20] *Auto-vectorization in GCC - GNU Project - Free Software Foundation (FSF)*. URL: https://gcc.gnu.org/projects/tree-ssa/vectorization.html (besucht am 27. 08. 2020).

[21] *Programming Guidelines for Vectorization*. Juli 2020. URL: https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/optimization-and-programming-guide/vectorization/automatic-vectorization/programming-guidelines-for-vectorization.html (besucht am 25. 07. 2020).