Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Optimierung eines Quantencomputer-Emulators

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Johannes Leonhard Maximilian von Stoephasius
geboren am: [redacted]
geboren in: [redacted]

Gutachter/innen: Prof. Dr. Jens-Peter Redlich
Prof. Dr. Henning Meyerhenke

eingereicht am: ..... verteidigt am: .....

Abstract

Diese Bachelorarbeit handelt von der Implementierung eines Quantencomputer-Emulators. Ziel ist die Optimierung aller Komponenten des Emulators für eine maximal mögliche Anzahl emulierbarer Qubits, als auch die geringste mögliche Laufzeit. Erreicht werden diese Ziele unter anderem durch die Entwicklung verschiedener Algorithmen für alle Komponenten, indem die mathematischen Grundlagen einzelner Teile untersucht werden. Code-Optimierungen werden beispielsweise durch Parallelisierung, Verwendung moderner Sprach-Features, oder durch die Verwendung von SIMD und anderen intrinsischen Funktionen umgesetzt. Intern wird die Simulation einer Superposition erreicht, indem auf einem Vektor operiert wird, welcher die Wahrscheinlichkeiten der Grundzustände repräsentiert. Das Ergebnis ist ein Quantencomputer-Emulator, welcher funktionsvollständig zu anderen Emulatoren ist und Quantenalgorithmen in vergleichbarer Laufzeit ausführen kann. Er erlaubt es Nutzern, beliebige eigene Quantenalgorithmen zu entwickeln und zu debuggen, sowie Programme von anderen Quantencomputer-Emulatoren nachzubauen.

Inhaltsverzeichnis

1 Motivation

Quantencomputer erlauben die Lösung bestimmter Probleme in geringerer Zeitkomplexität, als es mit einem klassischen Computer möglich ist. Da jedoch erst im letzten Jahr zum ersten Mal mehr als 50 Qubits miteinander verschränkt wurden [3], wird es noch einige Jahre dauern, bis Quantencomputer für Konsumenten verfügbar sind. Abhilfe schafft die Emulation eines Quantencomputers, mit dessen Hilfe Quantenalgorithmen auch ohne einen Quantencomputer ausgeführt werden können. Produkte wie Microsofts *Q#* oder Googles *Cirq* erlauben ein leichtes Erlernen der Technologie, bieten jedoch keine einfache Möglichkeit zu erlernen, wie Teile des Emulators intern operieren. So hat der *Q#*-Compiler eine Länge von über 53.000 Zeilen ohne Dokumentation über die Struktur des Projekts und *Cirq* eine Länge von über 100.000 Zeilen Quellcode. Kleinere Projekte, welche die Implementierung eines Quantencomputer-Emulators demonstrieren, existieren auf Plattformen, wie GitHub oder GitLab. Jedoch beschäftigen sich diese Projekte lediglich mit Machbarkeitsbeweisen und verwenden für Teile des Emulators die mathematisch trivialen Algorithmen, welche, wie gezeigt wird, sowohl eine sehr schlechte Laufzeit haben, als auch die Anzahl der emulierbaren Qubits erheblich einschränken. Projekte, die sowohl eine möglichst geringe Laufzeit und hohe Anzahl emulierbarer Qubits als Ziel haben und dabei leicht versteh- und modifizierbar sind, existieren nicht.

2 Zielsetzung

Die Lücke zwischen Machbarkeitsbeweisen und komplexen Bibliotheken soll von dieser Implementierung geschlossen werden. Ziel ist, die Anzahl der emulierbaren verschränkten Qubits zu maximieren und gleichzeitig die Laufzeit optimierter Quantenalgorithmen zu minimieren. Weiter soll der Emulator funktionsvollständig zu anderen Emulatoren sein. Das heißt, dass Programme von anderen Emulatoren direkt in Programme für den entwickelten Emulator übersetzt werden können.
Der Endnutzer soll weiterhin die Möglichkeiten haben, Code für den Quantenalgorithmus mit herkömmlichem Code zu mischen, beispielsweise um Konditionen einzubauen oder leicht Operationen auf mehrere Qubits anzuwenden. Weiter soll der innere Zustand des Emulators keine Black-Box sein; um das Debuggen von Algorithmen zu erleichtern, sollte das Design des Emulators es ermöglichen, dass der Endnutzer leicht die nötigen Rückschlüsse ziehen kann, weshalb Operationen einen gewissen Einfluss auf das System nehmen.
Die Zielplattform soll dabei ein moderner Desktop-PC mit nur einer CPU sein, wobei der Prozessor mindestens aus der 4. Intel Core Generation, oder der AMD Excavator Familie stammt. Auf die Verwendung von Bibliotheken, mit der Ausnahme der *C Standard Library*, *OpenMP* und *libm* wurde verzichtet, da der Fokus der Arbeit auf Optimierungen, und nicht der Wahl der richtigen Bibliothek, liegt.
Da bereits Quantencomputer-Emulatoren existieren, ist der Vergleich der Laufzeit von Quantenalgorithmen eine gute Metrik, um die Effizienz einer Implementierung festzustellen. Ziel ist es, dass die Laufzeit der finalen Implementierung im selben Größenbereich liegt, wie die von anderen Emulatoren.

3 Methodik

Implementiert wurde der Emulator im GNU-Dialekt von C23. Kompiliert wurden alle Programme, wenn nicht anders spezifiziert, mit GCC 13.2.1 und den Flags `-march=native -O3 -mavx2 -fopenmp -std=gnu2x -lm`. Durchgeführt wurden Benchmarks auf einem Desktop-PC mit 32 GB Hauptspeicher und Intel *i7-12700KF* (12 Kerne, 20 Threads) sowie einem Server mit 1 TB Hauptspeicher und 2 *Xeon Gold 6354* (jeweils 18 Kerne, 36 Threads). Auf beiden Maschinen ist eine 64-Bit GNU/Linux Distribution mit mindestens der Kernel-Version 5.14 installiert.
Für den Benchmark einzelner Funktionen wurden für jeden Datenpunkt mindestens 20 Messungen durchgeführt. Lag die durchschnittliche Laufzeit der ersten 20 Messungen unter einer Sekunde, so wurden 80 weitere Messungen durchgeführt. Sowohl die höchsten als auch die niedrigsten 5% wurden entfernt. Von den Verbleibenden 90% wurde der Durchschnitt gebildet, um den Wert für einen Datenpunkt zu ermitteln. Vor jedem Benchmark wurde überprüft, dass die durchschnittliche CPU-Auslastung pro Kern unter 7% liegt. Angegebene Laufzeiten beziehen sich, wenn nicht anders angegeben, stets auf die Real- und nicht auf die CPU-Zeit.
Um gleichzeitig auf mehren Zahlen zu operieren, wird unter anderem die x86 Befehlssatzerweiterung AVX2 verwendet, welche 16 265-Bit Register zu Verfügung stellt. Dies erlaubt eine parallele Ausführung bestimmter Befehle auf 4 `double`'s. Der Nachfolger von AVX2, AVX-512, wird nicht berücksichtigt, obwohl dieser die Größe der Register auf 512 Bit erhöht und damit die Anzahl der möglichen Datenpunkte, auf denen operiert werden kann, verdoppelt. Grund dafür ist, dass AVX-512 ab der 12. Generation von Intel Core Prozessoren nicht mehr unterstützt wird, da deren neue E-Cores AVX-512 nicht unterstützen. Da unklar ist, wann AVX-512 wieder relevant für moderne Desktop-CPUs ist, wird es im Folgenden nicht verwendet.

4 Vektor- und Matrix-Implementierung

4.1 Optimierungsmöglichkeiten

Repräsentiert wird der Zustand des Quantencomputer-Emulators durch einen Vektor, womit die Optimierung von Vektor- und Matrix-Operationen, sowie die Repräsentierung von Vektoren und Matrizen im Hauptspeicher Grundvoraussetzung für eine niedrige Laufzeit ist. Die Repräsentierung im Speicher ist dabei ausschlaggebend, wie weit andere Optimierungen möglich sind. Beispielsweise kann die Anordnung von Skalaren Einfluss darauf nehmen, wie gut mit Hilfe von SIMD auf mehreren Datenpunkten parallel operiert werden kann.
Basierend darauf werden die benötigten Vektor- und Matrix-Operationen einzelnen optimiert, indem zuerst mögliche Algorithmen verglichen und diese einzeln implementiert und mit folgenden Techniken optimiert werden. So wird versucht, sowohl mittels SIMD, als auch mittels klassischer Parallelisierung auf mehreren Datenpunkten parallel zu operieren. Weiter wird versucht, die Lokalität der verwendeten Daten zu erhöhen, um das Auftreten von Cache-Misses zu minimieren.

4.2 Repräsentation im Speicher

C unterstützt seit C99 die Datentypen `double complex` und `float complex`, welche die einfache Verwendung komplexer Zahlen mit der *C Standard Library* erlaubt. Die wichtigsten Operationen, wie Addition, Multiplikation, Konjugation, etc. werden vom Datentyp unterstützt, ebenso wie der schnelle Zugriff auf den Real- als auch den Imaginär-Teil. Eine Möglichkeit ist die Implementierung eines Vektors, beziehungsweise einer Matrix, als Array von `double complex`. Die Alternative ist die Implementierung des Vektors beziehungsweise der Matrix als 2 Arrays von `double`'s, das Erste für den Real- und das Zweite für den Imaginär-Teil.

```
1 struct Vector {
2     int size;
3     double complex* data;
4 };
```

```
1 struct Vector {
2     int size;
3     double complex* real;
4     double complex* imag;
5 };
```

Vorteil der Verwendung des `double complex` Typs sind dessen optimierte Operationen. Für Skalare erreicht `double complex` die bestmöglichen Laufzeiten. Beim Versuch, Matrix-Oprationenen mit dieser Matrix-Repräsentation zu implementieren, wurde das Problem offensichtlich, dass `double complex` die Verwendung von SIMD-Intrinsics massiv erschwert. Da sich stets Real- und Imaginär-Anteil abwechseln, ist eine Isolierung der beiden so langsam, dass die Verwendung von SIMD mit AVX2 keine Zeitvorteile bringt. Auch wenn AVX2 theoretisch einfache Operationen um einen Faktor von bis zu 4 beschleunigen kann, ist dies in der Praxis durch das Layout der Daten limitiert. Wenn die zu ladenden Daten nicht linear im Speicher liegen, kann der Overhead vom richtigen Laden der Daten schnell größer sein, als die gesparte Zeit. Dies tritt vor allem dann auf, wenn nur wenige Operationen auf den Daten durchgeführt werden und sich so die Zeit zum Laden der Daten nicht amortisieren kann.
Weiter benötigen nicht alle Quantenalgorithmen komplexe Zustände und verwenden lediglich den Real-Teil. Bei der Verwendung von 2 getrennten Arrays für Imaginär- und Real-Anteil benötigt die Umwandlung eines rein reellen Vektors in einen komplexen Vektor lediglich

eine Allokation. Bei der Verwendung eines Arrays von double complex muss hingegen über das komplette Array iteriert werden.

4.3 Allokation

Die Allokationen von Matrizen bzw. Vektoren hat auf die Laufzeit von Operationen, besonders bei kleinen Matrizen bzw. Vektoren, prozentual einen großen Einfluss. Diesen zu minimieren ist besonders wichtig, da der im Folgende untersuchte Strassen-Algorithmus zur Matrix-Multiplikation eine hohe Anzahl von Allokationen benötigt. Sowie *malloc* als auch *calloc* benötigen einen Kernel-Call, welche eine Vielzahl von Taktzyklen benötigen, was besonders bei oft auftretenden Allokationen einen massiven Einfluss auf die Laufzeit nehmen kann. Eine mögliche Alternative kann *alloca* sein, eine Funktion, die Speicher als Stack-Frame und nicht wie üblich auf dem Heap allokiert. So muss *alloca* nur den Stack-Pointer modifizieren, um beliebig viel Speicher zu allokieren, ein Kernel-Call ist nicht nötig. Das hat jedoch den Nachteil, dass beispielsweise eine aufgerufene Funktion nicht Speicher mit *alloca* allokieren und dann die Adresse der aufrufenden Funktion zurückgeben kann. Grund dafür ist, dass der allokierte Speicher jederzeit überschrieben werden kann. Das erschwert das Schreiben von Funktionen, die beispielsweise eine Matrix erstellen, da diese nicht die Allokation durchführen können. Das macht die Verwendung von Makros oder *forced inline* Funktionen nötig, um Konstruktor-Funktionen zu Schreiben. *VLA*s (variable length arrays) haben gegenüber *alloca* den Nachteil, dass ihre Lifetime abhängig vom Scope ist, in dem sie definiert wurden. Folglich kann mit ihnen ein Konstruktor nur als Makro implementiert werden, da selbst eine *inlined* Konstruktor-Funktion out-of-scope geht. *Alloca* hingegen geht erst dann out-of-scope, wenn die Funktion, die *alloca* aufgerufen hat, returnt.

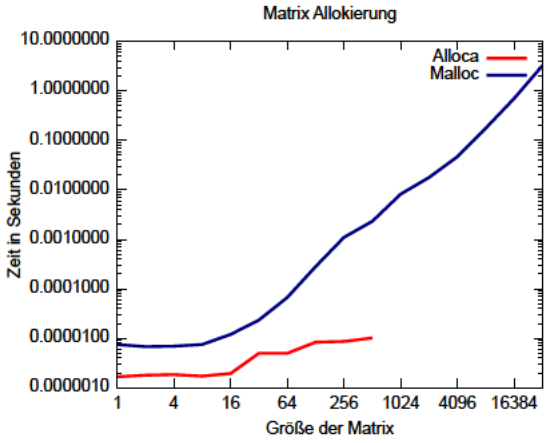

Abbildung 1: Vergleich der Laufzeit, um eine Matrix zu allokieren.

Wie erwartet resultiert die Verwendung von *alloca* in einer wesentlich geringeren Laufzeit. So ist die Stack-Allokation bei einer Matrix der Größe 512×512 um einen Faktor von über 100 schneller. Jedoch hat *alloca* den großen Nachteil, dass es beim Auftreten eines Fehlers nicht nullptr zurückgeben wird, sondern die Adresse, an der der allokierte Speicher hätte sein

sollen. Wird also die maximale Stack-Größe überschritten, kann dieser Fehler nicht gehandelt werden und kann so zu einer Segmentation Fault führen.

4.4 Matrix-Multiplikation

Auch wenn der iterative Algorithmus zur Multiplikation von 2 Matrizen bekannt ist, ist dieser klassische Algorithmus nicht ideal. Werden 2 Matrizen der Größe $n \times n$ miteinander mit dem iterativen Algorithmus multipliziert, so ist die Zeitkomplexität $\mathcal{O}(n) = n^3$. Strassens Algorithmus zur Matrix-Multiplikation, welcher mittels *Divide and Conquer* arbeitet, hat jedoch lediglich eine Zeitkomplexität von $\mathcal{O}(n) = n^{\log_2 7} \approx n^{2,807}$ [14]. Das Problem des optimalen Algorithmus zur Matrix-Multiplikation ist immer noch ungelöst. Der aktuell beste praktikable Algorithmus hat eine Zeitkomplexität von $\mathcal{O}(n) = n^{2,37286}$ [2], während ein Algorithmus mit der minimal besseren Zeitkomplexität $\mathcal{O}(n) = n^{2,371866}$ [4] eine so hohe Laufzeit hat, dass er keine Praxis-Relevanz besitzt. Das Projekt AlphaTensor arbeitete ebenfalls an dem Problem und versuchte mittels künstlicher Intelligenz neue Algorithmen zu finden. Da die Verwendung dieser Algorithmen sehr komplex ist, werden im Folgenden nur die Laufzeiten vom iterativen und vom Strassen-Algorithmus untersucht. Der Strassen-Algorithmus weist die Besonderheit auf, dass aufgrund der Teilberechnungen viele Allokationen notwendig sind, weshalb die Verwendung von *alloca* deutlichen Einfluss auf die Performance nehmen kann.

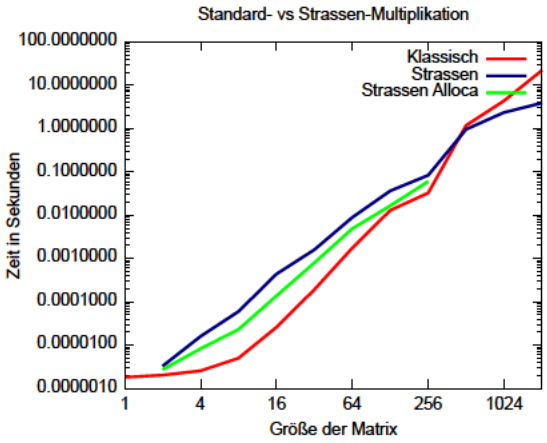

Abbildung 2: Für große Matrizen ist Strassens Algorithmus besser, für Kleine der Iterative.

Wie erwartet ist der Strassen-Algorithmus für große Matrizen aufgrund der besseren Laufzeitkomplexität schneller als der iterative Algorithmus, bei kleinen Matrizen ist jedoch der iterative Algorithmus schneller. Für jede Matrix-Größe ist die Implementierung des Strassen-Algorithmus, die *alloca* verwendet, schneller als die Implementierung, die *malloc* und *calloc* verwendet. Jedoch trat bereits ab einer Matrix-Größe von 512×512 ein Stack-Overflow auf, was die Verwendung von *alloca* in dieser Situation nicht möglich macht. Da *alloca* bei Allokations-Fehlern keine Form des Feedbacks gibt, kann, abhängig von der Verwendung des Stacks vor Aufruf des Strassen-Algorithmus mit *alloca*, das Programm durch einen Overflow terminieren. Das ist, trotz gutem Laufzeit-Vorteil nicht vertretbar.

Eine einfache Optimierung der Implementierung der iterativen Matrix-Multiplikation ist die Umordnung der Schleifen, mit denen über die Matrizen iteriert wird. Die intuitive Lösung iteriert so über die Matrizen, wie man sie meist auch schriftlich multiplizieren würde.

```
1  void multiply(Matrix const* restrict const a, Matrix const* restrict const b, Matrix*
       restrict const c) {
2      for (int i = 0; i < a->n; i++) {
3          for (int j = 0; j < a->n; j++) {
4              for (int k = 0; k < a->n; k++) {
5                  c->real[i * a->n + j] += (a->real[i * a->n+ k] * b->real[k * a->n + j
                       ]) - (a->imag[i * a->n+ k] * b->imag[k * a->n + j]);
6                  c->imag[i * a->n + j] += (a->real[i * a->n+ k] * b->imag[k * a->n + j
                       ]) + (a->imag[i * a->n+ k] * b->real[k * a->n + j]);
7              }
8          }
9      }
10 }
```

Tauscht man jedoch den j- mit dem k-Loop, so ist das Cache-Alignment für die Matrizen a und c besser, nur noch für die b-Matrix muss bei jedem Schritt eine Zeile gesprungen werden [13].

```
1  void multiply_rearanged(Matrix const* restrict const a, Matrix const* restrict const
       b, Matrix* restrict const c) {
2      for (int i = 0; i < a->n; i++) {
3          for (int k = 0; k < a->n; k++) {
4              for (int j = 0; j < a->n; j++) {
5                  c->real[i * a->n + j] += (a->real[i * a->n + k] * b->real[k * a->n +
                       j]) - (a->imag[i * a->n + k] * b->imag[k * a->n + j]);
6                  c->imag[i * a->n + j] += (a->real[i * a->n + k] * b->imag[k * a->n +
                       j]) + (a->imag[i * a->n + k] * b->real[k * a->n + j]);
7              }
8          }
9      }
10 }
```

Um das Cache-Alignment weiter zu verbessern ist es ebenfalls möglich, die Matrizen in mehrere Chunks zu unterteilen, über die man iteriert, was Cache-Misses reduzieren, aber selbst einen eigenen Overhead erzeugen kann.

```
1  void multiply_chunked(Matrix const* restrict const a, Matrix const* restrict const b,
       Matrix* restrict const c) {
2      const int chunkSize = 1024;
3      for(int jChunk = 0; jChunk < a->n; jChunk += chunkSize){
4          for (int i = 0; i < a->n; i++) {
5              for (int kChunk = 0; kChunk < a->n; kChunk += chunkSize) {
6                  for (int k = 0; k < chunkSize; k++) {
7                      for(int j = 0; j < chunkSize; j++){
8                          c->real[i * a->n + jChunk + j] += (a->real[i * a->n + kChunk
                               + k] * b->real[kChunk * a->n + k * a->n + jChunk + j]) -
                               (a->imag[i * a->n + kChunk + k] * b->imag[kChunk * a->n +
                                k * a->n + jChunk + j]);
9                          c->imag[i * a->n + jChunk + j] += (a->real[i * a->n + kChunk
                               + k] * b->imag[kChunk * a->n + k * a->n + jChunk + j]) +
                               (a->imag[i * a->n + kChunk + k] * b->real[kChunk * a->n +
                                k * a->n + jChunk + j]);
10                     }
11                 }
12
13             }
14         }
15     }
16 }
```

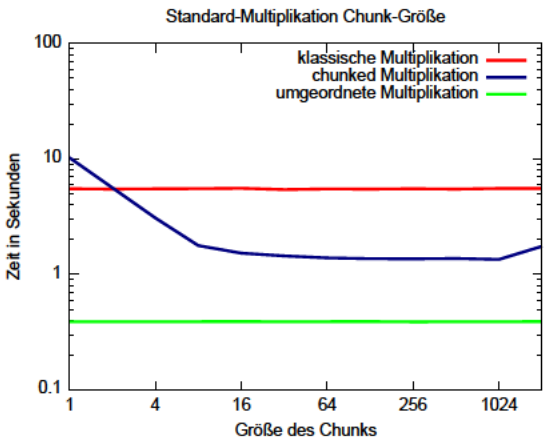

Abbildung 3: Gechunkte Multiplikation einer 2048×2048 Matrix.

Wie erwartet sorgt das Tauschen der Loops für eine deutlich verbesserte Performance, das Aufteilen der Matrix in Chunks sorgt jedoch für einen so großen Overhead, dass bei keiner der getesteten Matrix- und Chunk-Größen ein Performancegewinn möglich war. Weiter sorgte auch die Transformation einer der Matrizen a oder b, was das Cache-Alignment verbessert hätte, nicht für eine bessere Laufzeit.
Dieser optimiere iterative Algorithmus zur Matrix-Multiplikation kann genutzt werden, um die Laufzeit des Strassen-Algorithmus zur Matrix-Multiplikation zu verbessern. Auch wenn der itertative Algorithmus eine höhere Laufzeitkomplexität hat, so ist er bei kleinen Matrizen schneller. Strassens Algorithmus hat jedoch den Vorteil, dass es irrelevant ist, wie das Teilproblem der Matrix-Multiplikation gelöst wird. Der typische Rekursionsanker ist die Multiplikation von 2 Matrizen der Größe 2×2, dieser kann jedoch auf jede beliebige Matrixgröße gesetzt werden. Damit muss nur festgestellt werden, bei welcher Matrix-Größe

der Algorithmus-Wechsel am besten ist.

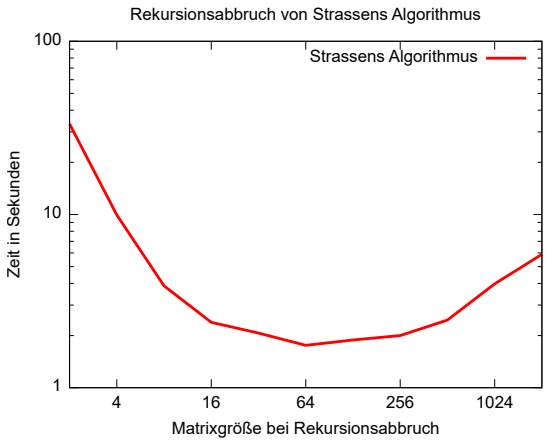

Abbildung 4: Multiplikation von 2 2048×2048 Matrizen mit verschiedenen Rekursionsankern.

Der frühere Abbruch der Rekursion verbessert wie vermutet die Laufzeit erheblich, mit der zum Testen verwendeten Hardware ist das ideale Ende der Rekursion bei einer Matrix-Größe von 64×64 erreicht.
Die Parallelisierung von Strassens Algorithmus zur Matrix-Multiplikation kann leicht umgesetzt werden. Da der Algorithmus mittels *Divide and Conquer* arbeitet, kann jede der Teilberechnungen einzeln durchgeführt werden, wobei nur das Zusammensetzen der Teile nicht immer parallelisierbar ist. Im Falle von Strassens Algorithmus müssen 8 Matrix-Produkte berechnet werden, welche anschließend nur addiert und in die finale Matrix kopiert werden müssen. Bei jedem Rekursionsschritt kann damit die Anzahl der möglichen Threads verachtfacht werden. Da der Algorithmus stets neue Matrizen allokieren muss, um das Ergebnis von Teilberechnungen zu speichern, liegen die vom Algorithmus verwendeten Daten meist nicht linear im Speicher. Umso höher die Anzahl der Threads ist, desto öfter müssen Daten erneut in den Cache geladen werden, weshalb ein Abbruch der Parallelisierung, ebenso wie der Abbruch der Rekursion, sinnvoll ist.

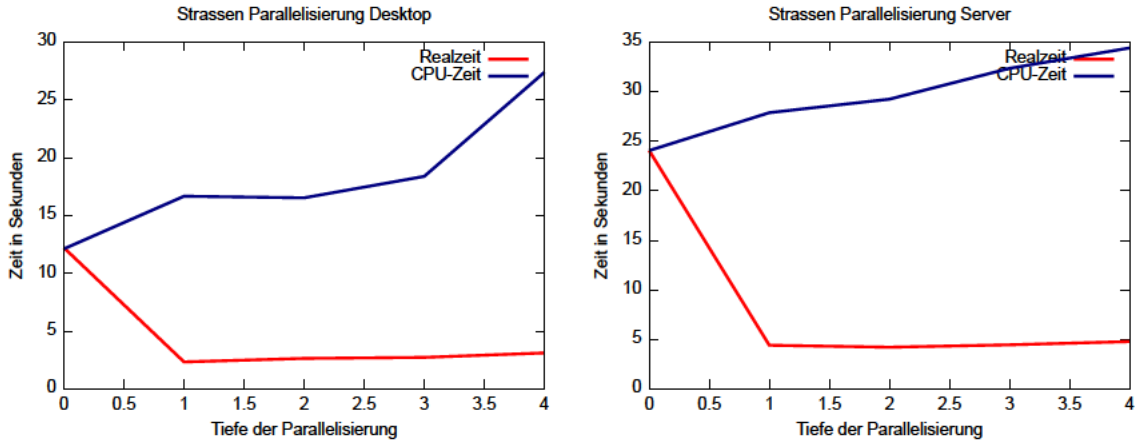

Abbildung 5: Vergleich von Parallelisierungs-Tiefen auf Desktop und Server.

Wie erwartet bringt sowohl auf dem Desktop-PC als auch auf dem Server die Parallelisierung der Tiefe 1 bereits einen enormen Vorteil für die Realzeit, ohne die CPU-Zeit massiv zu beeinflussen. Wird die Parallelisierungs-Tiefe auf 2 gesetzt, so ergibt sich auf dem Server noch eine minimale Reallaufzeit-Verbesserung, während auf dem Desktop-PC die Reallaufzeit bereits wieder ansteigt. Auch wenn geringe CPU-Zeit nicht das Ziel der Optimierungen ist, ist dennoch die Parallelisierungs-Tiefe von 1 optimal für den Server. Da die Reallaufzeit mit der Parallelisierungs-Tiefe von 1 nur 5% höher ist, aber dafür die Anzahl der verwendeten Kerne nur ein Achtel ist, amortisiert sich die höhere Reallaufzeit durch die geringere CPU-Temperatur und den damit höheren möglichen CPU-Takt.

4.5 Kronecker-Produkt

Um zustands-beschreibende Vektoren oder transformations-beschreibende Matrizen zu bilden, wird das Kronecker-Produkt benötigt. Dieses liefert als Ergebnis alle möglichen Produkte der Skalare von 2 Matrizen.

$$M = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \otimes \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \neq \begin{bmatrix} a * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} & b * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \\ c * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} & d * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \\ e * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} & f * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \end{bmatrix}$$

$$\neq \begin{bmatrix} a & 2a & 3a & b & 2b & 3b \\ 4a & 5a & 6a & 4b & 5b & 6b \\ c & 2c & 3c & d & 2d & 3d \\ 4c & 5c & 6c & 4d & 5d & 6d \\ e & 2e & 3e & f & 2f & 3f \\ 4e & 5e & 6e & 4f & 5f & 6f \end{bmatrix} = M$$

Die Ungleichungen sind wichtig, da die Matrix von Matrizen nur eine visuelle Repräsentation der Berechnung ist, aber keine mathematische Gleichheit besteht.
Da bei der Berechnung stets alle möglichen Kombinationen der Skalare beider Matrizen

betrachtet werden müssen, ist die Laufzeitkomplexität bei der Berechnung des Kronecker-Produkts der $n \times n$ Matrix N und der $m \times m$ Matrix M stets $\mathcal{O}(n, m) = n^2 m^2$. Damit ist bei der Optimierung besonders die Minimierung von Cache-Misses das Ziel.
Die erste Möglichkeit der Implementierung ist es, linear über alle Indizes der Ergebnis-Matrix zu iterieren und die entsprechenden Skalare aus N und M zu bestimmen. Da dadurch lediglich linear über die Ergebnis-Matrix iteriert wird, kann es sowohl bei der Bestimmung des Skalars von N als auch M zu einem Cache-Miss kommen. Iteriert man hingegen über die Matrizen N und M, so bleibt der Skalar von N konstant, während über M linear iteriert wird. So treten gehäufte Cache-Misses nur in der Ergebnis-Matrix auf.

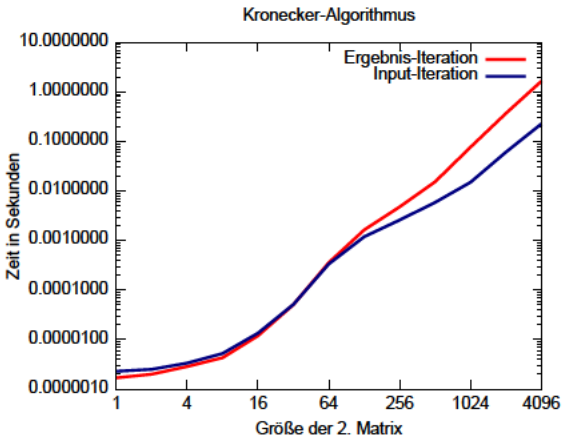

Abbildung 6: Laufzeit der Kronecker-Produkt-Berechnung von 2 Matrizen.

Aufgrund der besseren Cache-Lokalität ist die Iteration über die Eingabe-Matrizen für große Matrizen um einen Faktor von über 5 schneller.

5 Emulator-Komponenten

5.1 Funktionsweise

Im Gegensatz zu einem normalen Computer kann ein Quantencomputer bzw. ein Quantencomputer-Emulator nicht nur auf einem Zustand operieren, sondern auf mehren Zuständen gleichzeitig. Das ermöglicht es Quantenalgorithmen, bestimmte Probleme auf einem Quantencomputer mit geringerer Zeitkomplexität zu lösen, als es mit einem Computer möglich ist. Beispielsweise ist die Fraktionierung eines Integers, welche ein **NP**-Problem ist, mit Shor's Algorithmus hingegen nur ein **BQP**-Problem. Dabei beschreibt **BQP** die Komplexitätsklasse der Probleme, die auf einem Quantencomputer in Polynomialzeit lösbar sind, wobei die Fehlerrate unter 1/3 liegt. Das Verhältnis von **NP** zu **BQP** ist nicht bekannt [10]. Jedoch ist ein Quantencomputer nicht mächtiger als ein Computer; er ist ebenfalls Turing-Vollständig, das heißt, dass ein Quantencomputer eine Turingmaschine abbilden kann, ebenso wie eine Turingmaschine einen Quantencomputer abbilden kann. Eine Emulation eines optimalen Quantenalgorithmus liegt logisch stets in derselben oder einer höheren Komplexitätsklasse als der beste klassische Algorithmus, da sonst die Emulation des Quantenalgorithmus der beste klassische Algorithmus wäre, und folglich so in der selben Komplexitätsklasse wie er selbst liegt.
Da ein Quantencomputer nicht nur auf einem Zustand operieren kann ist die Anzahl der Zustände maximal gleich der Anzahl aller möglichen Zustände; im Kontext von Bits heißt das, dass ein Quantencomputer auf allen möglichen Bit-Kombinationen gleichzeitig operieren kann. Ist n die Anzahl der Bits, so liegt damit die maximale Anzahl der Zustände bei 2^n.
Repräsentiert werden kann dies auf mehrere Arten und Weisen, die effizienteste ist die Darstellung in Form eines Vektors. Dabei beschreiben die enthaltenen Skalare die Wahrscheinlichkeit, dass bei einer Messung aller Qubits in der Pauli-Z Basis, das simulierte System in den jeweiligen Zustand übergeht [6].
Beispielsweise existieren bei 3 Qubits 8 mögliche Zustände, in welche das System übergehen kann, wobei folgender Vektor eine Repräsentation ist:

$$\begin{bmatrix}\alpha\\ \beta\\ \gamma\\ \delta\\ \epsilon\\ \zeta\\ \eta\\ \theta\end{bmatrix}, \text{ wobei: } \begin{matrix}|\alpha^2| = \mathcal{P}(000)\\ |\beta^2| = \mathcal{P}(001)\\ |\gamma^2| = \mathcal{P}(010)\\ |\delta^2| = \mathcal{P}(011)\\ |\epsilon^2| = \mathcal{P}(100)\\ |\zeta^2| = \mathcal{P}(101)\\ |\eta^2| = \mathcal{P}(110)\\ |\theta^2| = \mathcal{P}(111)\end{matrix}, \text{ wenn der Vektor ein Einheitsvektor ist, also: } \left|\begin{bmatrix}\alpha\\ \beta\\ \gamma\\ \delta\\ \epsilon\\ \zeta\\ \eta\\ \theta\end{bmatrix}\right| = 1$$

Die im Vektor enthaltenen Skalare sind komplexe Zahlen, weshalb stets der Betrag des Quadrats genommen werden muss, um die Wahrscheinlichkeit zu erhalten.
Modifiziert werden kann der Zustand eines Quantencomputers durch die Anwendung von Gates, welche (bis auf die Messung) stets unitäre Transformationen, als umkehrbare Operationen sind [15]. Die Definition des Quantenalgorithmus ist analog zu der des Algorithmus. Es handelt sich um eine endliche Abfolge von Gates, die auf den Zustand eines Quantencomputers angewendet werden, welche terminiert, aber nicht deterministisch sein muss. Ein einfacher Algorithmus ist die Nachbildung eines Münzwurfs, wofür ein einzelnes Qubit mit dem *Ha-*

damard-Gate (kurz H-Gate) in die Superposition gebracht wird. Die Wahrscheinlichkeit, dass dieses Qubit bei einer Messung zu 1 oder 0 springt, liegt danach bei jeweils 50%, die Messung zerstört diese Superposition und das Qubit springt in einen der beiden Zustände. Dieser Quantenalgorithmus ist nicht deterministisch, da es physikalisch nicht möglich ist vorherzusagen, in welchen Zustand das System übergeht, praktisch ist ein Emulator dennoch stets deterministisch, wenn er lediglich Pseudo-Zufallszahlen verwendet.
Ein weiteres Beispiel für einen Quantenalgorithmus ist die Erstellung eines Bell-Zustands, welcher die Verschränkung von zwei Qubits demonstriert. Beide Qubits sind zu Beginn nicht verschränkt und im Zustand 0, $|00\rangle$ in Bra-Ket-Notation. Auf das Erste wird wie beim Münzwurf das H-Gate angewendet, wodurch das System in den Zustand $\frac{|00\rangle+|10\rangle}{\sqrt{2}}$ übergeht, bei einer Messung sind die beiden Zustände $|00\rangle$ und $|10\rangle$ gleich wahrscheinlich. Das *controlled not*-Gate (auch CNOT- oder CX-Gate genannt) flippt das Target-Bit genau dann, wenn das Control-Bit 1 ist; wendet man dieses auf den aktuellen Zustand an, wobei das erste Qubit das Control-Bit und das zweite das Target-Bit ist, so geht das System in den Zustand $\frac{|00\rangle+|11\rangle}{2}$ über, da das zweite Qubit nur dann geflipt wird, wenn auch das erste Qubit im Zustand 1 ist. Folglich hängt der Zustand des zweiten Qubits stets vom Zustand des ersten Qubits ab und umgekehrt, wird eines der beiden gemessen, so springt auch das andere in denselben Zustand.

5.2 Input

Die Berechnung des Inputs hat es zur Aufgabe, aus einer Abfolge von Bits bzw. einer Repräsentation von Bits, einen Vektor, im folgenden Register, zu bilden, der eben jener Abfolge von Bits bzw. Bit-Repräsentationen entspricht. Mathematisch betrachtet heißt das, dass jedem auf 0 gesetztem Bit der Vektor $\begin{bmatrix}1\\0\end{bmatrix}$ und jedem auf 1 gesetztem Bit der Vektor $\begin{bmatrix}0\\1\end{bmatrix}$ zugeordnet wird [12]. Das Kronecker-Produkt dieser Vektoren ist der Vektor, welcher diese Kombination an Bits beschreibt. Beispielsweise gilt für folgende Kombination von Qubits:

$$|101\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix}$$

Alternativ ist es ebenfalls möglich, bei der Erstellung des Registers alle Bits auf 0 zu setzen, und danach auf alle entsprechenden Bits das *not*-Gate (kurz X-Gate) anzuwenden. Das erlaubt eine sehr effiziente Erstellung des Registers, da lediglich ein auf 0 gesetztes Register allokiert werden und das erste Element auf 1 gesetzt werden muss. Wie im Unterabschnitt Gates gezeigt wird, hat jedoch die Anwendung von Gates eine höhere Zeitkomplexität und ist in jeder Situation langsamer.
Der Algorithmus, welcher den mathematisch trivialen Ansatz verwendet ist simpel, so berechnet er nacheinander die Kronecker-Produkte der Vektoren, die den Bits entsprechen.

```
Function calculate_input(states: Vector[]):
    current: Vector = states[0];
    for i ← 1 to states.length do
        current ← kronecker_product(left: current, right: states[i]);
    return current;
```

Algorithmus 1 : Triviale Input-Vektor-Berechnung.

Die Zeitkomplexität zur Berechnung des Kronecker-Produkts ist abhängig von der Anzahl der Elemente der beiden Vektoren, für das Kronecker-Produkt eines Vektors mit n Elementen mit einem Vektor mit m Elementen gilt: $\mathcal{O}(n, m) = n * m$. Da in diesem konkreten Fall der rechte Vektor stets 2 Elemente hat gilt: $\mathcal{O}(n) = n$. Da in der letzten Iteration der Schleife der Vektor *current* 2^{n-1} Elemente hat, lässt sich für alle berechneten Kronecker-Produkte die Zeitkomplexität mit $\mathcal{O}(n) = 2^n$ abschätzen, wobei n die Anzahl der Qubits ist. Die Zeitkomplexität des Algorithmus ist also $\mathcal{O}(n) = n * 2^n$.
Da das Kronecker-Produkt assoziativ ist, kann sowohl vom ersten zum letzten Vektor, als auch vom Letzten zum Ersten iteriert werden.

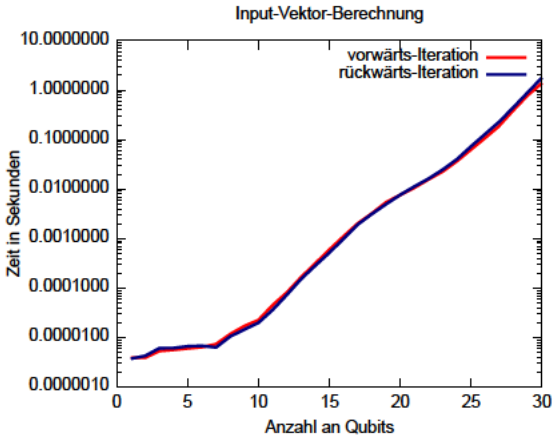

Abbildung 7: Die Reihenfolge der Iteration ist nur minimal relevant für die Laufzeit.

Die sehr schlechte Laufzeit ist nicht nur ein Ergebnis der hohen Komplexität des Algorithmus, sondern auch der hohen Speicherauslastung geschuldet. Da in jedem Schleifendurchlauf ein neuer Vektor allokiert wird, ist die Anzahl langsamer System-Calls abhängig von der Anzahl der Qubits. Weiter muss in jedem Schleifendurchlauf über 3 Arrays iteriert werden, woraus eine hohe Belastung des Caches folgt. Der größte Nachteil ist jedoch die Tatsache, dass bis auf ein Element der *current*-Vektor stets nur aus Nullen besteht. Folglich sollte ein besserer Algorithmus, lediglich den Index der einzigen 1 berechnen und erst am Ende einen Vektor allokieren.
Die Funktionsweise dieses Algorithmus kann auf zwei Weisen motiviert werden. Betrachtet man beim simplen Algorithmus den Index der einzigen 1, so fällt auf, dass sich bei jeder Iteration der Index mindestens verdoppelt (ist der Index 0, so kann er auch nach einem weiteren Schleifendurchlauf 0 sein). Der Grund ist, dass für jede 0, die über der 1 im *current*-Vektor

steht, 2 Nullen im Ergebnis-Vektor eingefügt werden, da, $0 * \begin{bmatrix}0\\1\end{bmatrix} = 0 * \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$ gilt. Ist der im aktuellen Schleifendurchlauf hinzuzufügende Vektor $\begin{bmatrix}1\\0\end{bmatrix}$, so verdoppelt sich der Index der einzigen 1; ist der aktuelle Vektor $\begin{bmatrix}0\\1\end{bmatrix}$, so ist der neue Index das Doppelte des alten plus 1, da zusätzlich noch eine weitere 0 im Ergebnis-Vektor steht.

```
Function calculate_input_optimized(states: Vector[]):
    index: int ← 0;
    for i ← 0 to states.length do
        index ← index * 2;
        if states[i] == Vector(0, 1) then
            index ← index + 1;
    Vector current ← vector_of_zeros(size: 2^states.length);
    current[index] ← 1;
    return current;
```

Algorithmus 2 : Input-Vektor-Berechnung mit Multiplikation und Inkrementierung.

Dieser Algorithmus ist identisch zu der logischen Folgerung aus der Definition eines Registers. So beschreibt beispielsweise der erste Skalar stets die Wahrscheinlichkeit, dass alle Qubits als 0 gemessen werden und der zweite, das alle Qubits bis auf das letzte als 0 gemessen werden. Damit ist die Binärdarstellung des Index einer Wahrscheinlichkeit stets die Aneinanderreihung der Bits, die eben jene Wahrscheinlichkeit beschreibt. Folglich reicht es zur Berechnung des Index, alle Bits aneinanderzureihen. Dabei beginnt man mit dem Index 0 und shiftet ihn für jede Bit-Repräsentationen um 1 nach links und flippt das letzte Bit des Index, wenn die aktuelle Bit-Repräsentation der 1 entspricht. Dieser Bit-Shift ist identisch zur Multiplikation mit 2 der ersten Motivation und der Flip des letzten Bits ist das Äquivalent der Inkrementierung. Am Ende muss nur der Vektor allokiert werden und der Skalar am berechneten Index auf 1 gesetzt werden.
Die Laufzeit der Implementierung des Algorithmus ist in der Praxis primär von der Geschwindigkeit der Allokierung abhängig. Dabei ist die Laufzeitkomplexität einer Heap-Allokierung vom Zustand des Programms und allgemein des Computers abhängig. Ist jedoch noch eine ausreichend große Menge Hauptspeicher verfügbar, so kann man davon ausgehen, dass für die Zeitkomplexität einer Allokierung eines Speicherbereichs der Größe n, die Abschätzung $\mathcal{O}(n) = n$ gilt. Folglich ist damit die Laufzeitkomplexität des Algorithmus $\mathcal{O}(n) = 2^n$.

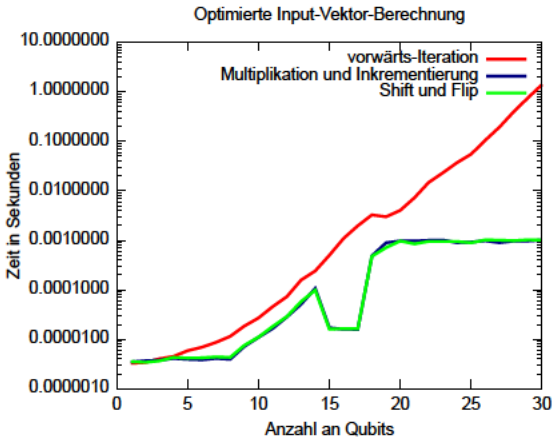

Abbildung 8: Vergleich der trivialen und algorithmisch optimierten Input-Vektor-Berechnung.

Ein Zeitunterschied zwischen der auf Multiplikation und der auf Shifts basierenden Implementierung war nicht zu erwarten und trat nicht auf. So wird sowohl der Bit-Shift als auch die Multiplikation vom Compiler durch eine Addition des Index mit sich selbst ersetzt. Auch wenn die Inkrementierung und der Bit-Flip nicht identisch übersetzt wurden, ist durch Instruction-Pipelining die Laufzeit identisch. Insgesamt sind damit beide Implementierungen wie erwartet schneller als der triviale Algorithmus. Der Unterschied in der Laufzeitkomplexität sorgt dafür, dass bei 28 Qubits der triviale Algorithmus um einen Faktor von 1000 langsamer ist.
Die Anomalie in beiden optimierten Algorithmen für 15, 16 und 17 Qubits konnte nicht komplett geklärt werden. Auch wenn sich mit jedem zusätzlichen Qubit die Menge an zu allokierendem Speicher verdoppelt und ein weiterer Schleifendurchlauf nötig ist, konnte stets für diese Anzahl an Qubits eine augenscheinlich zu niedrige Laufzeit beobachtet werden. Mögliche Ursprünge wie Cache Warm-Up oder Compile-Time-Optimizations konnten ausgeschlossen werden. Eine mögliche Erklärung wäre, dass von `malloc` ab 256 KB eine *transparent Hugepage* allokiert wird.
Die einzige Möglichkeit der Parallelisierung und Verwendung von SIMD lag beim trivialen Algorithmus in der Optimierung des Kronecker-Produkts, der optimierte Algorithmus lieferte keine Möglichkeiten der weiteren Optimierung der Implementierung.

5.3 Gates

Gates haben die Aufgabe der Manipulation des Zustands des Quantencomputers. Dabei werden in diesem Unterabschnitt nur Gates betrachtet, die lediglich auf einem einzelnen Qubit operieren. Controlled Gates, welche zusätzlich auch den Zustand anderer Qubits verwenden, werden im folgenden Unterabschnitt behandelt.
Mathematisch entspricht die Anwendung eines Gates auf einen Quantenzustand der Manipulation des Zustand-Vektors mit einer Transformations-Matrix. Diese Transformation ist aufgrund physikalischer Gegebenheiten stets unitär, also umkehrbar, und wird durch eben genannte Matrix beschrieben. Gebildet wird diese Matrix mit Hilfe kleinerer Matrizen, identisch zur Berechnung des Input-Registers. So wird für alle Qubits des Registers eine

beliebige unitäre 2×2 Matrix gewählt und das Kronecker-Produkt dieser Matrizen gebildet. Für Qubits, die nicht modifiziert werden sollen, wird die 2×2 Identitäts-Matrix gewählt, für die Anderen eine beliebige unitäre 2×2 Matrix. In einer Emulation kann diese Matrix stets beliebig sein, bei Quantencomputern steht jedoch oft lediglich nur eine limitierte Anzahl verschiedener Gates zur Verfügung.
Die triviale Implementierung der Berechnung einer Matrix, die ein Gate auf ein einzelnes Qubit anwendet, ist damit folglich lediglich die Berechnung des Kronecker-Produkts von einer Folge von Identitäts-Matrizen, einer beliebigen Matrix und einer weiteren Folge von Indentitäts-Matrizen [8].

```
Function calculate_gate_matrix(qubit_count: int, target_qubit: int, gate:
 Matrix):
    current: Matrix ← Matrix(1);
    for i ← 0 to qubit_count do
        if i == target_qubit then
            current ← kronecker_product(left: current, right: gate);
        else
            current ← kronecker_product(left: current, right: identity_matrix(n:
             2));
    return current;
```

Algorithmus 3 : Triviale Gate-Matrix-Berechnung.

Die Zeitkomplexität hängt nur von der Anzahl der Qubits ab, mit welcher die Größe der resultierenden Matrix bestimmt werden kann. So hat die Matrix *current* bei n Qubits nach der Schleife eine Größe von $2^n \times 2^n$, also einen Speicherverbrauch von $\mathcal{O}(n) = (2^n)^2 = n^{2n}$. Da die Schleife n Durchläufe benötigt und sowie die Allokierung des neuen *current* als auch die Berechnung des Kronecker-Produkts eine Zeitkomplexität von $\mathcal{O}(n) = n^{2n}$ hat, ist damit die Laufzeitkomplexität der gesamten Funktion $\mathcal{O}(n) = n * n^{2n}$. Da am Ende jeder Schleife das alte *current* gelöscht werden kann, liegt die Speicherkomplexität der gesamten Funktion bei nur $\mathcal{O}(n) = n^{2n}$.
Da bei diesem Algorithmus primär die Kronecker-Produkte von Identitäts-Matrizen gebildet werden, ist eine mögliche Verbesserung, diese zu überspringen. Da das Kronecker-Produkt von 2 Identitäts-Matrizen wieder eine Identitäts-Matrix ist, können alle Matrizen vor und hinter dem Gate zusammengefasst werden, sodass nur 2 Kronecker-Produkte berechnet werden müssen.

Function `calculate_gate_matrix_optimized`(*qubit_count: int, target_qubit: int, gate: Matrix*)**:**

left_mat: Matrix ← identity_matrix(n: 2^{target_qubit});
right_mat: Matrix ← identity_matrix(n: $2^{qubit\_count-target\_qubit-1}$);
return kronecker_product(left: kronecker_product(left: left_mat, right: gate), right: right_mat);

Algorithmus 4 : Optimierte Gate-Matrix-Berechnung.

Bei beiden berechneten Kronecker-Produkten ist die Laufzeitkomplexität $\mathcal{O}(n) = n^{2n}$, womit auch die Laufzeitkomplexität der gesamten Funktion bei $\mathcal{O}(n) = n^{2n}$ liegt. Da jedoch in diesem Fall über 3 große, und nicht wie davor über 2 große und eine kleine Matrix iteriert wird, kann durch ein häufigeres Auftauchen von Cache-Misses die Geschwindigkeit stark beeinträchtigt werden.

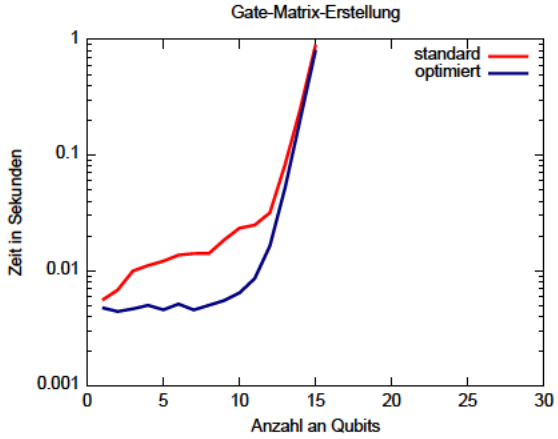

Abbildung 9: Trotz besserer Zeitkomplexität ist der optimierte Algorithmus nicht wesentlich schneller.

Auch wenn beide Algorithmen eine sehr ähnliche Performance haben, fällt besonders auf, dass beide schon bei nur 15 Qubits eine Sekunde und 12 GB Hauptspeicher zur Berechnung der Matrix benötigen, sodass auf den meisten Desktop-PCs nicht mehr als 16 Qubits emulierbar sind.
Die Anwendung der Matrix auf ein Register ist trivial; so muss lediglich das Produkt von der Matrix und dem Vektor, welcher das Register repräsentiert, gebildet werden. Unter gewissen Umständen kann jedoch durch das Zusammenfassen von Matrizen trotz höherer Zeitkomplexität eine geringere Laufzeit erreicht werden. Da die Matrix-Multiplikation assoziativ ist, kann eine beliebige Anzahl von Matrizen zu einer Einzelnen zusammengefasst werden, was trotz der hohen Kosten der Matrix-Multiplikation Zeit sparen kann. So liegt die Zeitkomplexität um das Produkt von p Matrizen für n Qubits zu bilden zwar bei $\mathcal{O}(n, p) = p * 2^{n*\log_2 7}$, die Anwendung dieser Matrix hat jedoch nur eine Zeitkomplexität von $\mathcal{O}(n) = 2^{2n}$ und muss so nur ein einziges, statt p mal durchgeführt werden. Will man beispielsweise 2 Matrizen auf jede mögliche Input-Kombination anwenden, halbiert sich die Laufzeit für hohe n fast:

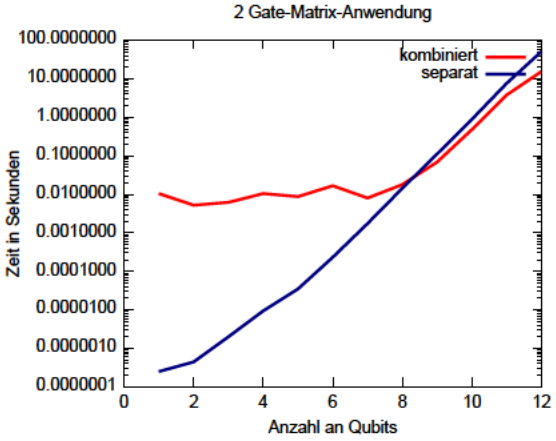

Abbildung 10: Bei häufiger Nutzung rentiert sich die Matrix-Multiplikation.

Das Problem des hohen Speicherbedarfs wird dadurch nicht gelöst, Ziel sollte also ein Algorithmus sein, welcher ohne die Erstellung einer Matrix auskommt. Dafür lässt sich eine Eigenschaft der transformations-beschreibenden Matrix ausnutzen, nämlich dass jede Zeile bis auf 2 Werte nur Nullen enthält. Beispielsweise gilt bei der Anwendung eines beliebigen Gates X auf das zweite von 3 Qubits:

$$X = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \text{ mit } a, b, c, d \in \mathbb{C}$$

$$M = I \otimes X \otimes I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} a & 0 & b & 0 & 0 & 0 & 0 & 0 \\ 0 & a & 0 & b & 0 & 0 & 0 & 0 \\ c & 0 & d & 0 & 0 & 0 & 0 & 0 \\ 0 & c & 0 & d & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & a & 0 & b & 0 \\ 0 & 0 & 0 & 0 & 0 & a & 0 & b \\ 0 & 0 & 0 & 0 & c & 0 & d & 0 \\ 0 & 0 & 0 & 0 & 0 & c & 0 & d \end{bmatrix}$$

Damit gilt für einen Vektor R, welcher ein Quantenregister mit 3 Qubits beschreibt:

$$R = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \\ \epsilon \\ \sigma \\ \eta \\ \theta \end{bmatrix}, \text{mit } \alpha, \beta, \gamma, \delta, \epsilon, \sigma\ \eta, \theta \in \mathbb{C}$$

$$M * R = \begin{bmatrix} \alpha * a + \gamma * b \\ \beta * a + \delta * b \\ \alpha * c + \gamma * d \\ \beta * c + \gamma * d \\ \epsilon * a + \eta * b \\ \sigma * a + \theta * b \\ \epsilon * c + \eta * d \\ \sigma * c + \theta * d \end{bmatrix}$$

Ersetzt man $\alpha, \beta, \gamma, ...$ durch den Index im Vektor in Binärform R, welcher gleich der Bit-Kombination ist, dessen Wahrscheinlichkeit der Wert repräsentiert, so erhält man folgenden Vektor:

$$M * R = \begin{bmatrix} \alpha * a + \gamma * b \\ \beta * a + \delta * b \\ \alpha * c + \gamma * d \\ \beta * c + \gamma * d \\ \epsilon * a + \eta * b \\ \sigma * a + \theta * b \\ \epsilon * c + \eta * d \\ \sigma * c + \theta * d \end{bmatrix} = \begin{bmatrix} R[000] * a + R[010] * b \\ R[001] * a + R[011] * b \\ R[000] * c + R[010] * d \\ R[001] * c + R[011] * d \\ R[100] * a + R[110] * b \\ R[101] * a + R[111] * b \\ R[100] * c + R[110] * d \\ R[101] * c + R[111] * d \end{bmatrix}$$

Es fällt auf, dass in jeder Zeile die beiden Indizes bis auf das mittlere Bit identisch sind, der Grund dafür liegt in der Funktionsweise eines Gates. So operiert es nur auf dem Target-Qubit, dass heißt, das für den neuen Wert an einem beliebigen Index sowohl der Wert an Position des Index, als auch der Wert an Position des Index mit invertierten Target-Bit benötigt wird. Sei i der aktuell betrachtete Index und o der Index mit invertiertem Target-Bit, so gilt: ist das Target-Bit beim aktuellen Index 0, so ist der neue Wert $a * R[i] + b * R[o]$, ist das Target-Bit 1, so ist der neue Wert $c * R[i] + d * R[o]$. Daraus lässt sich folgender Algorithmus bilden:

```
Function apply_gate(input: Vector, qubit_count: int, target_qubit: int, gate:
 Matrix):
    output: Vector ← vector_of_zeros(size: 2^qubit_count);
    target_bit_flip_mask: int ← 1 << ((qubit_count - target_qubit - 1));
    for i ← 0 to 2^qubit_count do
        other: int ← i ^ target_bit_flip_mask;
        if i < other then
            output[i] ← input[i] * gate[0, 0] + input[other] * gate[0, 1];
        else
            output[i] ← input[i] * gate[1, 1] + input[other] * gate[1, 0];
    return output;
```

Algorithmus 5 : Gate-Anwendung ohne Matrix.

Die Berechnung des Index mit invertiertem Target-Bit wird erreicht, indem eine Maske berechnet wird, welche nur an Position des Target-Bit eine 1 hat. XORt man diese mit dem Index, so erhält man den Index mit invertiertem Target-Bit. Berechneten kann man diese

Maske, indem man eine 1 so oft nach links shiftet, bis sie an der richtigen Position ist, bei n Qubits und dem Target-Index x sind also $n - x - 1$ Shifts erforderlich, da die Indizes beginnend von Null gezählt werden. Bei drei Qubits und dem zweiten, also mit Index 1, als Target, ist so ein einzelner Shift nach links erforderlich.
Da der Algorithmus bei n Qubits lediglich über alle 2^n Zeilen des Vektors iterieren muss und auch nur einmalig einen Vektor mit 2^n Elementen allokiert, ist die Laufzeitkomplexität als auch die Speicherkomplexität $\mathcal{O}(n) = 2^n$.
Dieser Algorithmus hat dennoch den Nachteil, dass er einen Output-Vektor allokiert, was dazu führen kann, dass die Anzahl maximal emulierbarer Qubits damit um eins verringert wird. Eine Optimierung des Algorithmus, sodass er *in-place* arbeitet, ist wünschenswert. Dafür muss lediglich der Inhalt des *else*-Zweigs in den if-Zweig verschoben werden und alle i und *other* miteinander getauscht werden.

Function `apply_gate(`*input: Vector, qubit_count: int, target_qubit: int, gate: Matrix*`):`

```
target_bit_flip_mask: int ← 1 << ((qubit_count - target_qubit - 1));
for i ← 0 to 2^qubit_count do
    other: int ← i ^ target_bit_flip_mask;
    if i < other then
        at_i: float ← input[i];
        at_other: float ← input[other];
        input[i] ← at_i * gate[0, 0] + at_other * gate[0, 1];
        input[other] ← at_i * gate[1, 0] + at_other * gate[1, 1];
```

Algorithmus 6 : Gate-Anwendung ohne Matrix und Allokierung.

Dieser Algorithmus hat dieselbe Laufzeitkomplexität, muss allerdings keinerlei Speicher allokieren.
Es fällt auf, dass bei der Hälfte aller Schleifendurchläufe die Bedingung $i < other$ nicht erfüllt ist und damit der aktuelle Durchlauf nicht notwendig ist. Ein Algorithmus, welcher nur über alle relevanten Paare von i und *other* iteriert, kann damit die Anzahl der Schleifendurchläufe halbieren. Lässt man das Target-Bit von allen i bei einer kompletten Schleife weg, so sind diese nur ein Zähler, welcher bei jedem Schleifendurchlauf inkrementiert wird. Umgekehrt kann man also einen Zähler verwenden und bei jedem Durchlauf an der richtigen Stelle eine 0 einfügen, um alle möglichen i zu erhalten. Aus diesen i können dann, wie bisher, die dazu gehörenden *other* berechnet werden. Das Einfügen einer 0 an der Stelle des Target-Bits ist leicht, so müssen zuerst alle Bits gespeichert werden, die rechts vom Target-Bit liegen. Dafür muss nur die Target-Bit-Maske um 1 dekrementiert werden, um eine Maske zu erhalten, die für alle Bits rechts vom Target-Bit 1 ist. Um alle Bits zu maskieren, die links vom Target-Bit sind, muss diese Maske lediglich invertiert werden. Mit diesen beiden Masken kann so ein Index in zwei Teile zerlegt werden, der Linke muss dann lediglich um 1 nach links geshiftet werden und danach beide Teile mit einem OR zusammengefügt werden.

```
Function apply_gate(input: Vector, qubit_count: int, target_qubit: int, gate: Matrix):
    target_bit_flip_mask: int ← 1 << (qubit_count - target_qubit - 1);
    right_bits_mask: int ← target_bit_flip_mask -1;
    left_bits_mask: int ← ∼ right_bits_mask;
    for i ← 0 to 2^qubit_count/2 do
        left_bits: int ← (i & left_bits_mask) << 1;
        right_bits: int ← i & right_bits_mask;
        combined: int ← left_bits | right_bits;
        other: int ← combined ^ target_bit_flip_mask;
        at_combined: float ← input[combined];
        at_other: float ← input[other];
        input[combined] ← at_combined * gate[0, 0] + at_other * gate[0, 1];
        input[other] ← at_combined * gate[1, 0] + at_other * gate[1, 1];
```

Algorithmus 7 : Gate-Anwendung ohne Matrix, ohne Allokierung und halbierter Iteration.

Wieder hat der Algorithmus eine Laufzeitkomplexität von $\mathcal{O}(n) = 2^n$ für n Qubits und benötigt keine Allokierung.
Zum Benchmarken der Gate-Anwendung ohne Matrix müssen alle möglichen Kombinationen implementiert werden, das heißt *out-of-place* mit voller Iteration, *in-place* mit voller Iteration, *out-of-place* mit halbierter Iteration und *in-place* mit halbierter Iteration. Jede diese Kombinationen wird mit und ohne SIMD-Intrinsics implementiert und zusätzlich auch in parallelisierter Form gebenchmarkt.
Die Implementierung ohne SIMD-Intrinsics basierend auf dem sehr C-ähnlichen Pseudocode und die Parallelisierung mit OpenMP ist einfach, Teile der SIMD-Intrinsics wiesen jedoch interessante Auffälligkeiten auf. So ist es für nur 2 der Kombinationen auf jeder getesteten Hardware schneller, nicht die Funktion _mm256_set_epi64x zu nutzen, um 4 Indizes in einem Register zu speichern. Stattdessen ist es nur für diese 2 Kombinationen schneller, die Indizes in ein Array mit 32-Bit-Alignment zu speichern und dieses am Stück mit _mm256_load_si256 in das Register zu kopieren. Für das Auslesen von AVX-Registern war es jedoch bei jeder Kombination auf jeder getesteten Hardware am schnellsten, die _mm256_storeu_si256-Funktion zu nutzen und die Werte in einem Array ohne Alignment zu speichern. Die finale Implementierung der *in-place* Implementierung mit halbierter Iteration, SIMD und Parallelisierung ist beispielsweise:

```
void apply_gate(ColumnVectorReal* restrict const input, MatrixReal const* restrict
    const gate, const uint target, const uint count) {
    const qint bits = (count - target - 1);
    const __m256i right_bits_mask = _mm256_set1_epi64x((1LL << bits) - 1LL);
    const __m256i left_bits_mask = _mm256_set1_epi64x(~((1LL << bits) - 1LL));
    const __m256i target_bit_mask = _mm256_set1_epi64x((1LL << bits));
#pragma omp parallel  for
    for (qint i = 0; i < input->n / 2; i += 4) {

        alignas(32)const qint indices [4] = {i, i+1, i+2, i+3};
        _mm_prefetch((const char *) indices, _MM_HINT_T0);
        const __m256i indices = _mm256_load_si256((const __m256i_u *) indices);
        const __m256i right_bits = _mm256_and_si256(indices, right_bits_mask);
        __m256i left_bits = _mm256_and_si256(indices, left_bits_mask);
        left_bits = _mm256_slli_epi64(left_bits, 1);
        const __m256i combined = _mm256_or_si256(left_bits, right_bits);
        const __m256i other = _mm256_xor_si256(combined, target_bit_mask);

        qint combined_array[4];
        qint other_array[4];

        _mm256_storeu_si256((__m256i_u*) combined_array, combined);
        _mm256_storeu_si256((__m256i_u*) other_array, other);

        for (int j = 0; j < 4; ++j) {
            const qint combined_index = combined_array[j];
            const qint other_index = other_array[j];
            const qfloat combined_value = input->real[combined_index];
            const qfloat other_value = input->real[other_index];

            input->real[combined_index] = combined_value * gate->real[0] +
                other_value * gate->real[1];
            input->real[other_index] = combined_value * gate->real[2] + other_value *
                 gate->real[3];
        }

    }
}
```

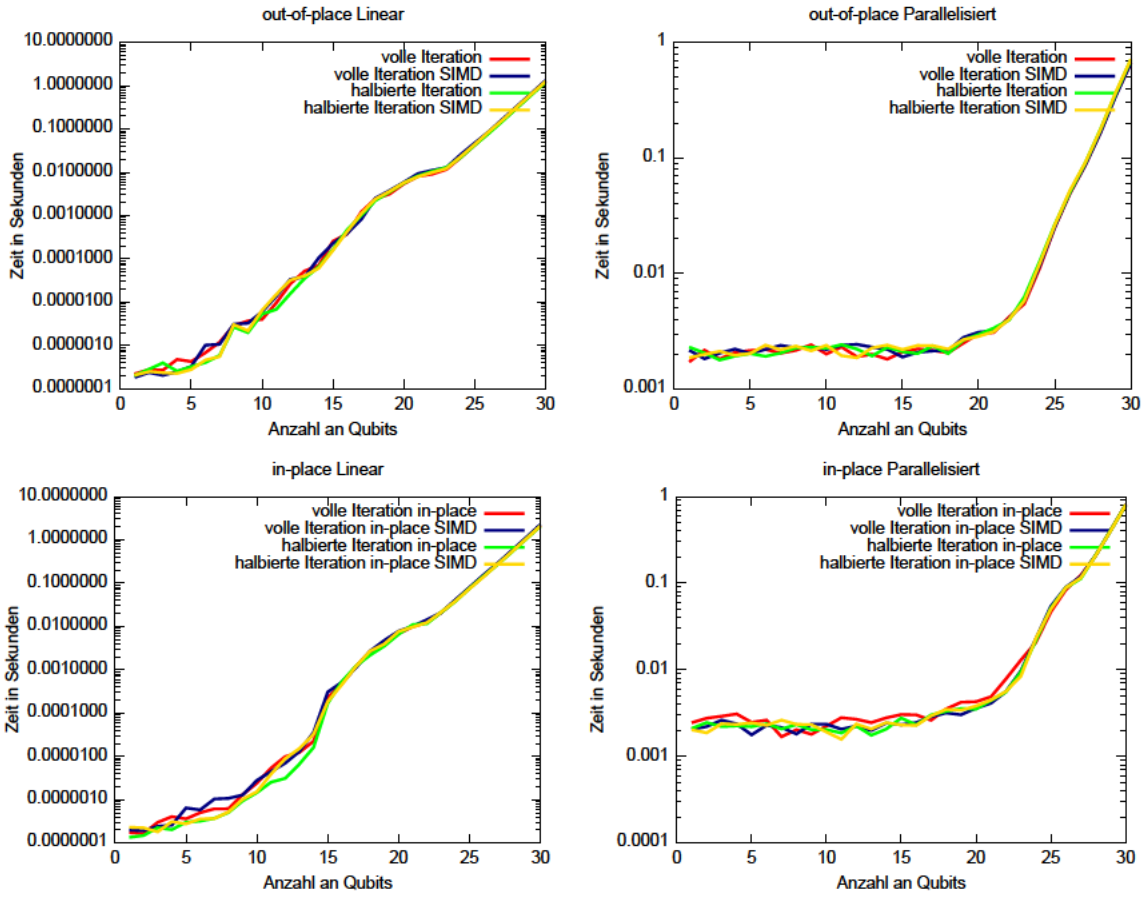

Abbildung 11: Vergleich der Performance aller Implementierungen.

Auch wenn die Unterschiede innerhalb jeder der Graphen nur gering ist, gab es stets Implementierungen, die die geringste Laufzeit hatten. Fasst man diese in ein einzelnes Diagramm zusammen erhält man:

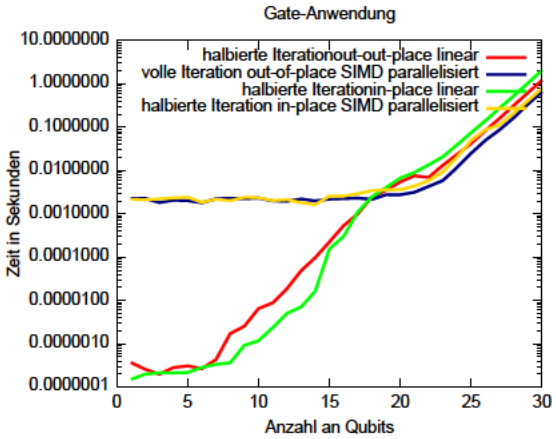

Abbildung 12: Vergleich der besten Implementierungen.

Damit kann endgültig der optimale Algorithmus gebildet werden, bei weniger als 17 Qubits ist die *in-place* Implementierung mit halbierter Iterierung ohne SIMD und Parallelisierung am schnellsten, bei mehr Qubits sollte die *out-of-place* Implementierung ohne halbierter Iterierung mit SIMD und Parallelisierung verwendet werden. Erst wenn die Allokierung des Output-Vektors nicht möglich ist, sollte die *in-place* Implementierung mit halbierter Iterierung, SIMD und Parallelisierung verwendet werden.
Dies ist jedoch nur der Algorithmus für rein reelle Wahrscheinlichkeiten, für komplexe Vektoren müssen alle Kombinationen erneut implementiert, optimiert und gebenchmarkt werden. Teilt man die Kombinationen wie beim reellen Vektor auf und wählt die beste Kombination, so erhält man:

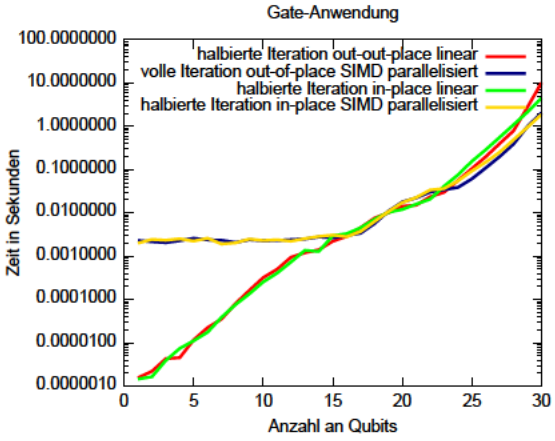

Abbildung 13: Vergleich der besten Implementierungen für komplexe Vektoren.

Auffällig ist die größere Überlappung der Performance aller Kombinationen von 16 bis 24 Qubits. Die Wahl der Kombinationen bleibt identisch, lediglich der Wechsel verschiebt sich von 17 zu 23 Qubits.

5.4 Controlled Gates

Controlled Gates erlauben die konditionale Anwendung von Gates. So wird ein Gate nur dann auf das Target-Qubit angewendet, wenn alle so genannten Control-Qubits im Zustand $|1\rangle$ sind. Das am meisten verwendete controlled Gate ist das *controlled not*-Gate (auch CX-, oder CNOT-Gate genannt), welches das Target-Qubit invertiert, wenn das einzelne Control-Qubit $|1\rangle$ ist. Werden zwei Qubits als Control verwendet, um ein Qubit zu invertieren, so nennt sich das Gate *controlled controlled not*-Gate (oder auch CCX-, CCNOT- oder Toffoli-Gate). Theoretisch kann jedes Gate, welches auf ein Qubit angewendet werden kann, mit beliebig vielen Control-Bits versehen werden, wobei in der Praxis jedoch meist Grenzen bei der Komplexität gesetzt sind.
Die algorithmische Umsetzung von controlled Gates erweist sich als schwieriger als die von gewöhnlichen Gates. So ist es nicht möglich, mittels Kronecker-Produkten von 2×2-Matrizen eine Transformations-Matrix zu bilden. Jedoch lässt sich der Algorithmus zur *Gate-Erstellung ohne Matrix* modifizieren, um eine Matrix zu erstellen. Für das *controlled not*-Gate mit beliebig vielen Control-Qubits beispielsweise muss lediglich über die Diagonale der Matrix

iteriert werden und falls die Control-Qubits nicht alle $|1\rangle$ sind, eine 1 eingetragen werden. Sind jedoch alle Control-Qubits $|1\rangle$, so wird für die Zeile das Target-Bit wie beim "Gate-Erstellung ohne Matrix"-Algorithmus geflippt. Der Test ob alle Conrol-Qubits $|1\rangle$ sind, kann ähnlich zum Flip des Target-Bits durchgeführt werden. So muss lediglich eine Maske erstellt werden, die für alle Control-Bits 1 ist. ANDet man diese Maske mit dem Index und erhält die Maske, so sind alle Control-Bits des Index 1.

```
Function create_cnot_matrix(qubit_count: int, target_qubit: int,
 control_qubits: int[]):
    control_bits_mask: int ← 0;
    for i ← 0 to control_qubits.length do
        control_bits_mask: int ← control_bits_mask | (1 << (qubit_count -
         target_qubit[i] - 1));
    target_bit_flip_mask: int ← 1 << (qubit_count - target_qubit - 1);
    output: Matrix ← matrix_of_zeros(size: qubit_count);
    for i ← 0 to qubit_count do
        if (i & control_mask) == control_mask then
            output[i ^ control_bits_flip_mask, i]← 1;
        else
            output[i, i]← 1;
    return output;
```

Algorithmus 8 : triviale CNOT-Erstellung.

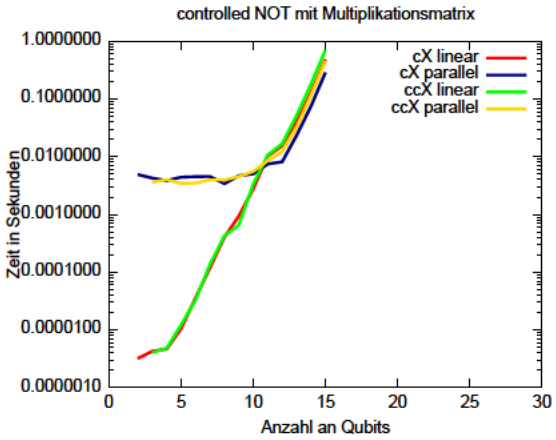

Abbildung 14: Laufzeitvergleich von Algorithmen zur Erstellung von cX-Matrizen.

Dieser Algorithmus hat, wie der für die triviale Gate-Matrix, eine Platzkomplexität von $\mathcal{O}(n) = 2^{2n}$ und benötigt ebenfalls eine Matrix-Multiplikation zur Ausführung. Damit ist dieser Ansatz es nicht wert, weiter optimiert zu werden. Jedoch kann durch eine kleine Modifikation ein wesentlich besserer Algorithmus erreicht werden. Da die Matrix zum Großteil

aus Nullen besteht, ist für jede Zeile die einzig relevante Information, ob die aktuelle Zeile modifiziert wird. Also muss für jeden Index, falls für ihn alle Control-Bits 1 sind, der Index mit invertiertem Target-Bit gespeichert werden [1].

```
Function create_controlled_indices(qubit_count: int, target_qubit: int,
 control_qubits: int[]):
    control_bits_mask: int ← 0;
    for i ← 0 to control_qubits.length do
        control_bits_mask: int ← control_bits_mask | (1 << (qubit_count -
          target_qubit[i] - 1));
    target_bit_flip_mask: int ← 1 << (qubit_count - target_qubit - 1);
    output: int[2^qubit_count];
    for i ← 0 to qubit_count do
        if (i & control_mask) == control_mask then
            output[i]← i ^ control_bits_flip_mask;
        else
            output[i]← -1;
    return output;
```

Algorithmus 9 : Controlled Gate Indizes-Vektor-Erstellung.

Dieser Vektor von Indizes erlaubt eine einfache Anwendung eines beliebigen Gates für diese exakte Konfiguration von Control- und Target-Qubits. Diese ist identisch zur Anwendung von Gates, nur dass zusätzlich überprüft werden muss, ob der Index -1 ist. Wie im Unterabschnitt Gates kann dieser Algorithmus sowohl *in-place* als auch *out-of-place* umgesetzt werden.

```
Function apply_controlled_gate_with_indices_vector(input: Vector,
 gate: Matrix, indices: int[]):
    output: Vector ← vector_of_zeros(size: input.length);
    for i ← 0 to input.length do
        other: int ← indices[i];
        if other ≠ -1 then
            if i < other then
                output[i] ← input[i] * gate[0, 0] + input[other] * gate[0, 1];
            else
                output[i] ← input[i] * gate[1, 1] + input[other] * gate[1, 0];
        else
            output[i]← input[i];
    return output;
```

Algorithmus 10 : Controlled Gate Indizes-Vektor-Anwendung.

Auch wenn die Zwischenspeicherung der Indizes nicht nötig ist, und stattdessen auch das Gate direkt angewendet werden kann, ergibt sich mit einer Modifikation des Algorithmus ein möglicher Performance-Vorteil. Da sich mit jedem zusätzlichen Control-Qubit die Anzahl der Indizes, die nicht -1 sind, halbiert, kann es vorteilhaft sein, lediglich die Indizes zu speichern, für die das Gate tatsächlich angewendet wird.

Function `create_cnot_reduced_indices(`*qubit_count: int, target_qubit: int, control_qubits: int[]*`):`

```
control_bits_mask: int ← 0;
for i ← 0 to control_qubits.length do
    control_bits_mask: int ← control_bits_mask | (1 << (qubit_count -
     target_qubit[i] - 1));
target_bit_flip_mask: int ← 1 << (qubit_count - target_qubit - 1);
output: Vector ← vector_of_zeros(size: (2^qubit_count) / (2^control_qubits.length));
index: int ← 0;
for i ← 0 to 1 << qubit_count do
    if (i & control_mask) == control_mask then
        other: int ← i ^ control_bits_flip_mask;
        if i < other then
            output[index * 2] ← i;
            output[index * 2 + 1] ← other;
            index ← index + 1;
return output;
```

Algorithmus 11 : reduzierte controlled Gate -Indizes-Vektor-Erstellung.

Bei der Implementierung der parallelisierten Variante ist jedoch die Existenz der *index*-Variable ein Problem. So müssen alle Threads sowohl lesend als auch schreibend auf sie zugreifen, was mit OpenMP auf den meisten Systemen einen Overhead zur Folge hat [5]. Da es sich jedoch um einen Integer handelt, kann ein *omp atomic increment*, statt eines *omp critical increment* verwendet werden, welches unter x86_64 vom Compiler optimiert werden kann. Dies geschieht, indem mit dem *LOCK*-Befehl-Präfix der Index direkt inkrementiert wird, statt auf eine kompliziertere Art und Weise den Index zwischen den Threads zu synchronisieren. Dies geschieht auf CPU-Ebene, indem sichergestellt wird, dass der Befehls-ausführende CPU-Kern exklusiven Zugriff auf die entsprechende Cache-Line hat [7].

```
1 if(i < other) {
2     qint index_copy;
3 #pragma omp atomic capture
4     index_copy = index++;
5     output[index_copy * 2]= i;
6     output[index_copy * 2 + 1]= other;
7 }
```

Auch wenn es möglich ist, die Position im *output*-Array direkt aus dem Index zu berechnen,

ist keine der Implementierungen schneller als ein zwischen den Threads synchronisier Index. Die Anwendung des resultierenden Index-Arrays ist beinahe identisch zur Anwendung des nicht resultierenden Index-Arrays. Der primäre Unterschied ist, dass bei der *out-of-place*-Variante zu Beginn der komplette Inhalt vom *input*- zum *output*-Vektor kopiert wird. Dies war in allen Fällen schneller als mögliche Alternativen, da dadurch die Hardware-optimierte Funktion *memcpy* genutzt werden kann. Besonders bei der $in-place$-Variante spart eine hohe Anzahl an Control-Qubits viel Zeit ein, da kein Speicher kopiert werden muss und die Anzahl der zu berechnenden Werte geringer ist. In der Praxis kommen jedoch in Algorithmen sehr selten controlled Gates mit mehr als zwei Control-Qubits vor, was den möglichen Vorteil bei der tatsächlichen Verwendung reduziert.
Zum benchmarken müssen erneut alle möglichen Kombinationen implementiert werden, dass heißt die Verwendung von einem kompletten, oder einem reduzierten Index-Vektor, *in-place* oder *out-of-place* und parallelisiert oder linear. Da die Wahl des Gates für das Target-Qubit für die Performance irrelevant ist, wird das CX- und das CCX-Gate verwendet. Um den möglichen Performance-Vorteil des Speicherns eines Index-Vektors zu testen, wird das Gate sowohl einmal als auch doppelt angewendet.

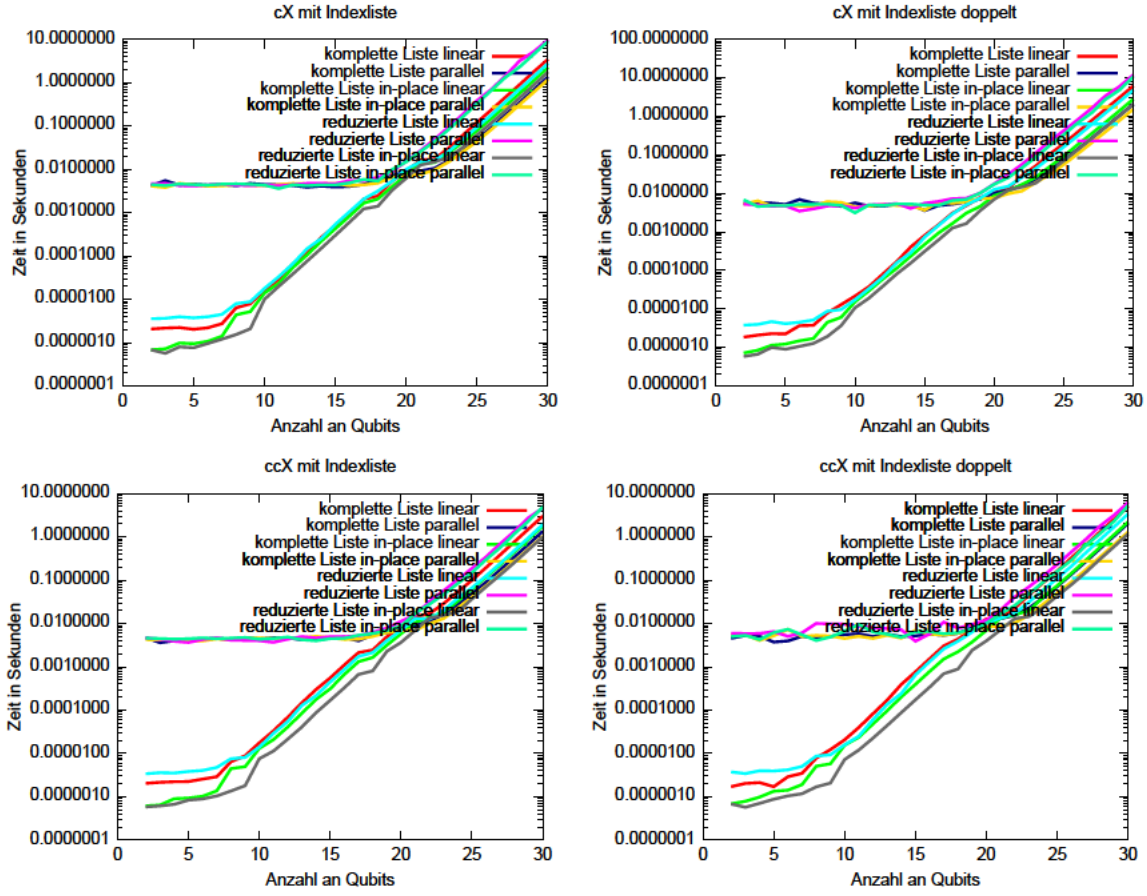

Abbildung 15: Performance-Vergleich aller Implementierungen mit Indexliste.

Wie erwartet sind für alle Implementationen Gates mit 2 Control-Qubits schneller als Gates

mit einem Control-Qubit. Die unter diesen Umständen am schnellsten arbeiteten Algorithmen werden nun mit dem Algorithmus verglichen, welcher keinen Index-Vektor zwischenspeichert.

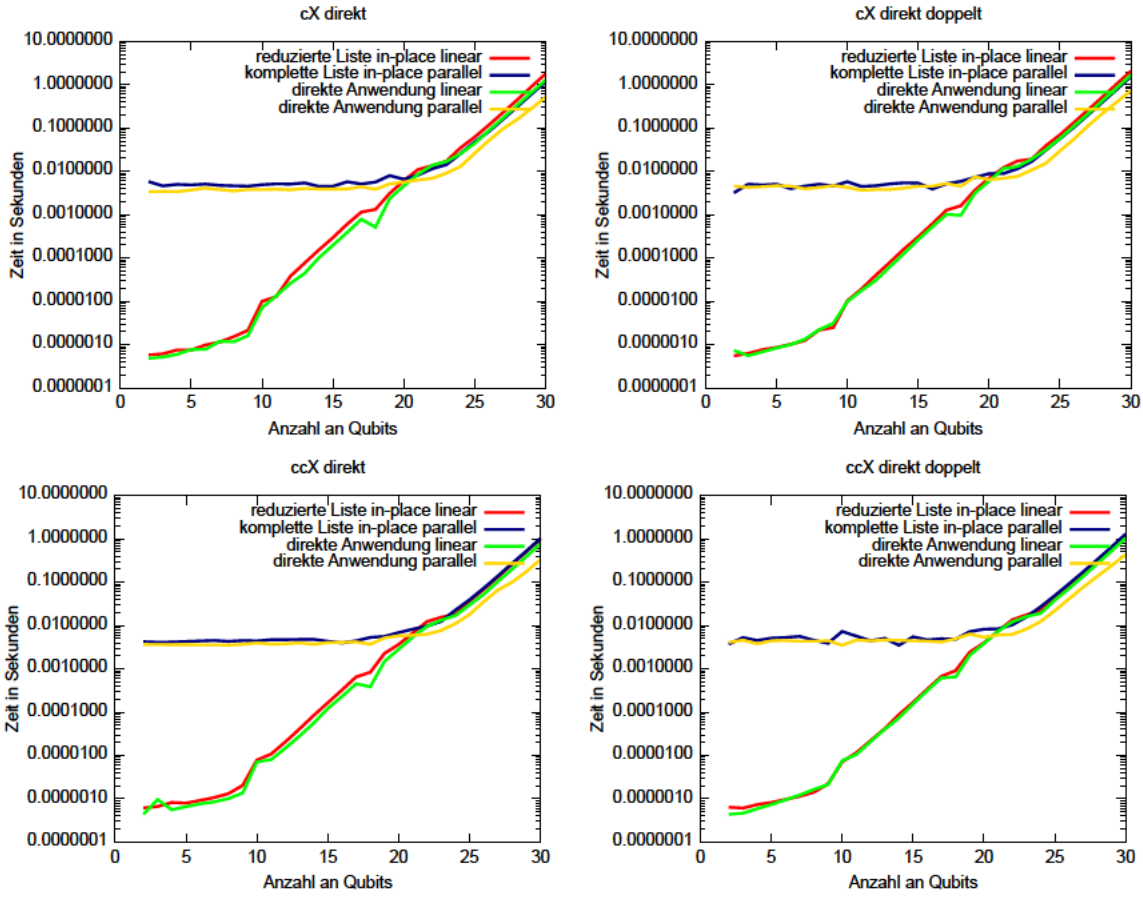

Abbildung 16: Performance-Vergleich aller Implementierungen ohne Indexliste.

In allen getesteten Situationen hatten die Algorithmen ohne gespeichertem Index-Vektor eine geringere Laufzeit. Theoretisch könnten Index-Vektoren bei einer sehr hohen Anzahl an Anwendungen eines spezifischen Gates effizienter sein, da diese jedoch in keinem gängigen Quanten-Algorithmus nötig sind, sind Index-Vektoren damit nicht praxisrelevant. Anzumerken ist, dass bei keinem der getesteten Algorithmen eine Verbesserung durch die Verwendung von SIMD-Intrinsics erreicht wurde.
Für komplexe Vektoren ist das Verhalten extrem ähnlich, jedoch verschiebt sicht der optimale Wechsel von der linearen zur parallelisierten Anwendung von 20 zu 21 Qubits.

5.5 Messungen

Die Messung von Qubits in einer Superposition ist ein irreversibler Prozess, welcher sie in den Zustand $|0\rangle$ oder $|1\rangle$ bringt. Dabei können entweder alle, ein Teil, oder auch nur ein einzelnes Qubit gemessen werden, wobei die Messung auch auf nicht gemessene Qubits Einfluss nehmen

kann. Betrachtet man beispielsweise den Bell-Zustand $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$, so führt eine Messung eines Qubits immer zum Kollaps der Superposition und bringt auch das andere Qubit in denselben Grundzustand $|0\rangle$ oder $|1\rangle$.
Da die Messung irreversibel ist und vom aktuellen Zustand abhängt, ist es nicht möglich, eine Matrix zu bilden, welche universell eine Messung beschreibt. Stattdessen muss bei jeder Durchführung einer Messung für jeden möglichen Grundzustand geprüft werden, wie hoch die Wahrscheinlichkeit ist, dass das System in eben jeden Zustand übergeht. Diese Wahrscheinlichkeit lässt sich aus dem Register berechnen, welcher das Quanten-System repräsentiert. Wie im Unterabschnitt Funktionsweise beschrieben, ist der Index einer Wahrscheinlichkeit gleich dem Grundzustand, welchen die Wahrscheinlichkeit repräsentiert. So beschreibt der Wert mit dem Index 0 die Wahrscheinlichkeit, dass das System in den Zustand übergeht, in welchem alle Qubits $|0\rangle$ sind und der Wert mit dem Index 3, dass alle bis auf die letzten beiden Qubits $|0\rangle$ sind.
Um aus einem Wert im Vektor die Wahrscheinlichkeit zu berechnen, muss lediglich der Betrag vom Quadrat des Werts berechnet werden. Für die Messung aller Qubits wählt man eine Zufallszahl zwischen 0 und 1 aus und iteriert über alle Werte im Vektor, während man die Wahrscheinlichkeiten addiert. Der erste Index, für welchen die Summe der Wahrscheinlichkeiten über der Zufallszahl liegt, ist der Zustand, in welchen das System übergeht. Dafür müssen alle anderen Wahrscheinlichkeiten und damit die Werte auf 0 gesetzt werden, und der Vektor anschließend normalisiert werden. Im Kontext der Messung aller Qubits bedeutet das, dass die Wahrscheinlichkeit des aktuellen Index auf 1 gesetzt wird, damit die Länge des Vektors wieder 1 ist.

```
Function measure_all(input: Vector, qubit_count: int):
    random: float ← random_between(0, 1);
    sum: float ← 0;
    for i ← 0 to 2^qubit_count do
        sum ← sum + abs(input[i]^2);
        if sum > random then
            input ← vector_of_zeros(size: 2^qubit_count);
            input[i] ← 1;
            return i;
```

Algorithmus 12 : Messung aller Qubits.

Aus dem zuzrückgebenen Index i kann, falls benötigt, ein Vektor von booleans berechnet werden, welcher den Zustand der Qubits beschreibt. Da das letzte Bit von i den Zustand des letzten Qubits beschreibt, kann von hinten nach vorne über alle Bits iteriert werden und immer das letzte Bit mit 1 verglichen werden, um den Zustand aller Qubits zu erhalten.

Function `get_states`(*index: int, qubit_count: int*)**:**

```
output: bool[qubit_count];
for i ← 0 to qubit_count do
    if index & 1 then
        output[qubit_count - 1 - i] ← true;
    else
        output[qubit_count - 1 - i] ← false;
    index ← index >> 1;
return output;
```

Algorithmus 13 : Berechnung aller Zustände.

Mit der Verwendung von SIMD können die Beträge der Quadrate von 4 Werten gleichzeitig berechnet werden, jedoch gibt es noch eine weitere Möglichkeit der Optimierung. Da lediglich nur ein einziges Mal der Fall betrachtet wird, dass die Summe größer als die Zufallszahl ist, kann durch die Reduktion der nötigen Additionen Zeit gespart werden. Addiert man die Beträge der Quadrate der Werte horizontal zusammen und addiert sie damit gemeinsam zur Summe, kann so die Anzahl aller Additionen gesenkt werden. Wenn der Fall auftritt, dass die Summe größer als die Zufallszahl ist, so kann die letzte Addition rückgängig gemacht werden, und die Werte einzeln addiert werden.

```
1  qint measure_all(const ColumnVectorReal *const restrict input) {
2      qfloat sum = 0;
3      const qfloat random = (qfloat) rand() / (qfloat) RAND_MAX;
4
5      for (qint i = 0; i < input->n; i += 4) {
6          const __m256d real = _mm256_loadu_pd(&(input->real[i]));
7          const __m256d squared = _mm256_mul_pd(real, real);
8          //calculate the sum
9          __m128d low = _mm256_castpd256_pd128(squared);
10         const __m128d high = _mm256_extractf128_pd(squared, 1);
11         low = _mm_add_pd(low, high);
12         const __m128d high64 = _mm_unpackhi_pd(low, low);
13         const double result = _mm_cvtsd_f64(_mm_add_sd(low, high64));
14
15         sum += result;
16         if (sum > random) {
17             sum -= result;
18             for (qint j = 0; j < 4; ++j) {
19                 sum += pow(input->real[i + j], 2);
20                 if (sum >= random) {
21                     memset(input->real, 0, sizeof(qfloat) * input->n);
22                     input->real[i + j] = 1; //normalize
23                     return i + j;
24                 }
25             }
26         }
27     }
28 }
```

Für die Umsetzung der horizontalen Addition existieren mehrere Möglichkeiten, wobei die Perfomance sehr stark von der verwendeten CPU abhängt [11]. So benötigt die Funktion *vextractf128* auf AMD CPUs vor *Zen 2* mindestens 8 Mikro-Befehle, während vergleichbare

Intel CPUs lediglich 3 Mikro-Befehle benötigen. Die verwendete Implementierung der Addition benötigt sowohl für Intel als auch für AMD insgesamt nur 4 Mikro-Operationen. Da der Fall, dass die Summe größer als die Zufallszahl ist, nur ein einziges Mal auftritt, war es stets schneller, die Beträge der Quadrate der Werte erneut zu berechnen, anstatt sie aus dem *squared*-Register auszulesen, da so das Register früher verfügbar wird.
Eine direkte Parallelisierung hätte den großen Nachteil, dass mehrere Threads auf die *sum*-Variable zugreifen müssen. Für eine effiziente parallelisierte Implementierung muss jeder Thread einzeln die Summe aller Wahrscheinlichkeiten bilden; anschließend können diese Summen der Threads zusammengerechnet werden. Wenn diese Summe größer als die Zufallszahl ist, kann wieder der letzte Summand abgezogen werden und alle zu dem Thread gehörenden Wahrscheinlichkeiten einzeln addiert werden, bis die Summe größer als die Zufallszahl ist.

```
1  qint measure_all_parallel(const ColumnVectorReal *const restrict input) {
2      const int threads = omp_get_max_threads();
3      qfloat sums[threads];
4  #pragma omp parallel for
5      for (int thread = 0; thread < threads; thread++)
6      {
7          sums[thread] = 0;
8
9          for (int i = thread; i < input->n; i += threads) {
10             sums[thread] += pow(input->real[i], 2);
11         }
12     }
13     qfloat random = (qfloat) rand() / (qfloat) RAND_MAX;
14     qfloat sum = 0;
15     for (int i = 0; i < threads; ++i) {
16         sum += sums[i];
17         if (sum > random) {
18             sum -= sums[i];
19             for (int j = i; j < input->n; j += threads) {
20                 sum += pow(input->real[j], 2);
21                 if (sum >= random) {
22                     memset(input->real, 0, sizeof(qfloat) * input->n);
23                     input->real[j] = 1;
24                     return j;
25                 }
26             }
27         }
28     }
29 }
```

Beim Benchmark werden die Implementierungen ohne SIMD, SIMD nur für die Berechnung der Wahrscheinlichkeiten und SIMD für die Berechnung der Wahrscheinlichkeiten und Summierung sowohl parallelisiert als auch nicht verglichen. Da alle Implementierungen dieselbe Laufzeitkomplexität von $\mathcal{O}(n) = 2^n$ haben, wobei n die Anzahl der Qubits ist, und bei keiner Implementierung Heap-Allokierungen notwendig sind, ist ein Performance-Unterschied nur zwischen linearen und parallelisierten Implementierungen erwartbar.

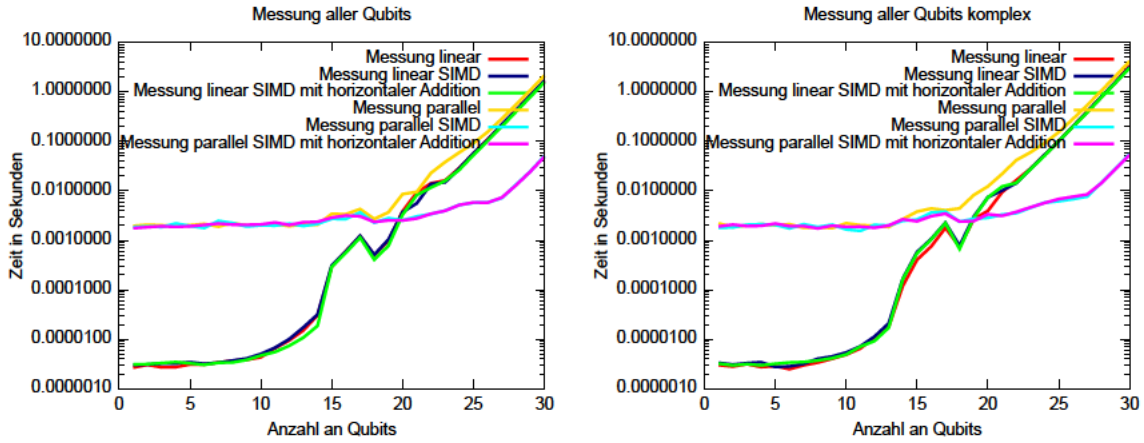

Abbildung 17: Laufzeitvergleich für Implementierungen für die Messung aller Qubits.

Für mehr als 20 Qubits ist die parallelisierte Implementierung mit SIMD am schnellsten, unabhängig davon, ob die Addition ebenfalls mit SIMD durchgeführt wurde. Dies gilt sowohl für komplexe, als auch rein reale Register. Für optimale Performance auf älteren AMD-CPUs ist damit die Implementierung mit Addition ohne SIMD besser geeignet. Für weniger als 20 Qubits ist die Wahl der Implementierung egal, solange sie nicht parallelisiert ist.
Neben der Messung aller Qubits ist es ebenfalls möglich, nur einen Teil der Qubits zu messen, wobei die meisten Algorithmen entweder alle oder nur ein einzelnes Qubit messen. Für die Messung eines einzelnen Qubits werden nicht die Wahrscheinlichkeiten aller Zustände betrachtet, sondern entweder die Wahrscheinlichkeiten der Zustände, bei dem das gemessene Qubit $|0\rangle$ ist, oder die Wahrscheinlichkeiten der Zustände, bei dem das gemessene Qubit $|1\rangle$ ist. Die Wahl, welche Menge an Wahrscheinlichkeiten man betrachtet, hat keinen Einfluss auf den Algorithmus, weshalb im Folgenden die Wahrscheinlichkeiten der Zustände verwendet werden, bei denen das gemessene Qubit $|0\rangle$ ist. Ist die Summe der Wahrscheinlichkeiten größer als eine Zufallszahl, so wird das gemessene Qubit als $|0\rangle$ gemessen, damit können alle Wahrscheinlichkeiten der Zustände, bei dem das gemessene Qubit $|1\rangle$ ist, auf 0 gesetzt werden. Anschließend muss der Vektor, der die Wahrscheinlichkeiten der Zustände beschreibt, wieder normalisiert werden. Dafür muss er nur durch seine Länge geteilt werden, welche die Wurzel Summe der verbliebenen Wahrscheinlichkeiten ist.
Für die Iteration über die Wahrscheinlichkeiten der Zustände, bei dem das gemessene Qubit $|0\rangle$ ist, gibt es mehrere Möglichkeiten, die auf bereits verwendeten Algorithmen aufbauen. Die erste Möglichkeit ist es, immer die für alle hintereinanderliegenden Werte im Vektor die Wahrscheinlichkeiten zu berechnen und danach die Werte zu überspringen, bei denen das gemessene Qubit $|1\rangle$ ist. Wird beispielsweise das zweit-letzte Qubit gemessen, werden abwechselnd für 2 Werte die Wahrscheinlichkeiten berechnet und danach 2 Werte übersprungen, da in der binär-Repräsentation des Werts das zweit-letzte Bit alle zwei Schritte wechselt. So kann über den Vektor iteriert und die Summe gebildet werden. Theoretisch könnte diese Iteration abgebrochen werden, sobald die Summe der Wahrscheinlichkeiten über der gewählten Zufallszahl liegt. Da jedoch die Länge des Vektors benötigt wird, welche aus der Summe aller Wahrscheinlichkeiten gebildet wird, muss mindestens einmal über alle Wahrscheinlichkeiten der Werte iteriert werden, bei dem das gemessene Qubit $|0\rangle$ ist [9].
Alternativ kann ebenfalls wieder eine Bit-Maske verwendet werden, um nur über die gewünschten

Zustände zu iterieren. Dabei muss jedoch nur die Variante dieses Algorithmus betrachtet werden, welche nur über die relevante Hälfte allter Bit-Kombinationen iteriert. Die Variante, die über alle Kombinationen iteriert, würde identisch zum ersten Algorithmus arbeiten, nur dass unnötigerweise über die irelevanten Zustände iteriert wird, bei denen das gemessene Qubit $|1\rangle$ ist.

```
Function measure_single(input: Vector, qubit_count: int, target_qubit: int):
    random: float ← random_between(0, 1);
    sum: float ← 0;
    step_size: int ← (1 << ((qubit_count - target_qubit) - 1));
    steps: int ← input.length / step_size ;
    index: int ← 0;
    for i ← 0 to steps do
        //iterate over states with target_qubit = |0⟩
        for u ← 0 to steps do
            sum ← sum + input[index]^2;
            index ← index + 1;
        index ← index + step_size; //skip states with target_qubit = |1⟩
    //set probabilities and normalize the vector:
    index ← 0;
    if sum > random then
        //measured Qubit is |0⟩
        length: float ← sqrt(sum);
        for i ← 0 to steps do
            //iterate over states with target_qubit = |0⟩
            for u ← 0 to steps do
                input[index] ← input[index] / length;
                index ← index + 1;
            //iterate over states with target_qubit = |1⟩
            for u ← 0 to steps do
                input[index] ← 0;
                index ← index + 1;
    else
        //measured Qubit is |1⟩
        length: float ← sqrt(1 - sum);
        for i ← 0 to steps do
            //iterate over states with target_qubit = |0⟩
            for u ← 0 to steps do
                input[index] ← 0;
                index ← index + 1;
            //iterate over states with target_qubit = |1⟩
            for u ← 0 to steps do
                input[index] ← input[index] / length;
                index ← index + 1;
```

Algorithmus 14 : Messung eines Qubits.

Dieser Algorithmus kann, wie erwähnt, modifiziert werden, um statt des *index*-Zählers eine

Bit-Maske zu verwenden. Beide Algorithmen können jeweils mit und ohne SIMD und mit und ohne Parallelisierung implementiert werden.

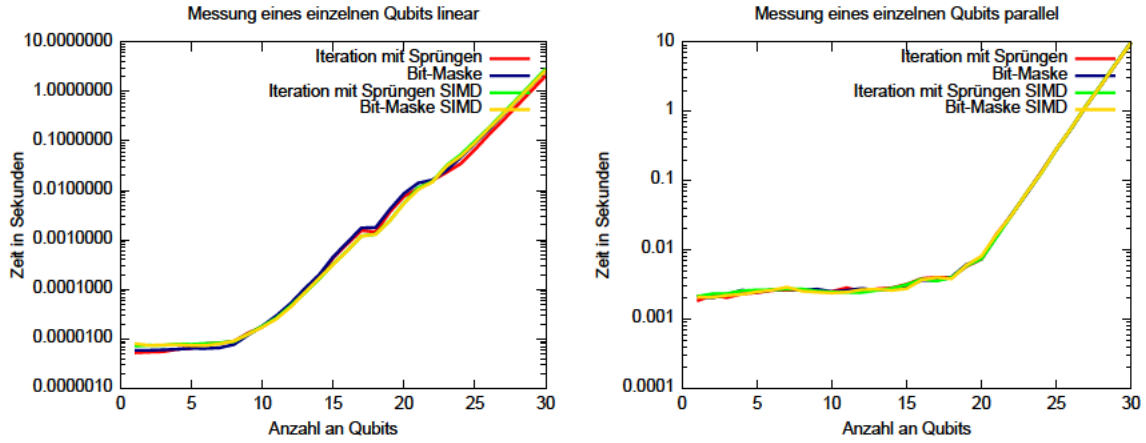

Abbildung 18: Laufzeitvergleich der Implementierungen für die Messung eines einzelnen Qubits.

Da alle Implementierungen bei beiden Tests fast identische Laufzeiten haben, ist nur die Anzahlt an Qubits relevant, bei der von einer linearen Implementierung zu einer parallelisierten gewechselt werden muss.

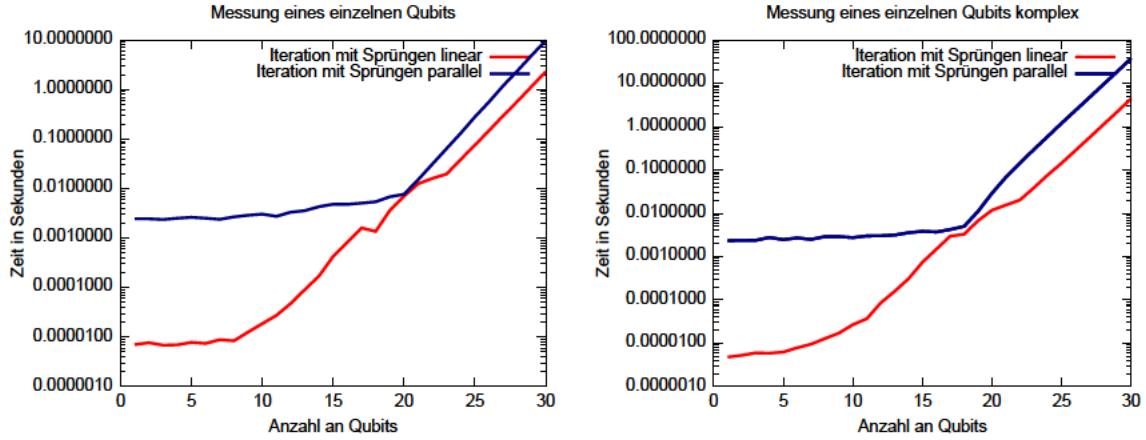

Abbildung 19: Vergleich von linearer und paralleler Implementierung für die Messung eines einzelnen Qubits.

Wie erwartet ist Laufzeit für wenige Qubits mit der parallelisierten Implementierung wesentlich höher, sehr auffällig ist allerdings, dass auf der getesteten Hardware für keine Anzahl an Qubits, mit der parallelisierten Implementierung eine bessere Laufzeit erreicht werden konnte. Ein Grund dafür könnte sein, dass mehrere Threads gleichzeitig schreibend auf die selbe Cache-Line zugreifen wollen, was einen Bottleneck zur Folge haben könnte.

6 Vergleich mit anderen Emulatoren

Zusammen ergeben alle beschriebenen Komponenten einen vollständigen Quantencomputer-Emulator, welcher beliebige Quantenalgorithmen ausführen kann. Dies erlaubt den Vergleich mit anderen Emulatoren, welche ebenfalls lokal ausgeführt werden können. Populäre Emulatoren sind Microsofts *Q#*, welches eine eigene in Programmiersprache ist, und Googles *Cirq* welches in Python implementiert ist.
Ein einfacher Algorithmus, welcher die Fähigkeiten eines Quantencomputers demonstriert, ist der Deutsch-Jozsa-Algorithmus. Dieser erhält als Eingabe ein Orakel, welches auf n Bits operiert. Dieses Orakel gibt entweder immer 0, 1, oder für genau die Hälfte aller Eingaben 0, bzw. 1 aus. Mit einem klassischen Computer müssen daher im Worst-Case $2^{n-1} + 1$ verschiedene Kombinationen dem Orakel übergeben werden, da davor nicht entschieden werden kann, ob das Orakel konstant (alle Eingaben werden zu einem Wert abgebildet) oder balanciert (die Hälfte der Eingaben wird zu einem Wert abgebildet) ist. Durch das Verschränken von Qubits ist es jedoch möglich, dass ein Quantencomputer mit nur einer einzelnen Eingabe entscheiden kann, ob das Orakel nur die Hälfte, oder alle Eingaben zu einem Wert abbildet. Der Deutsch-Jozsa-Algorithmus ist sehr gut als Benchmark geeignet, da er sowohl verschränkte, als auch unverschränkte Qubits verwendet.
Um unsortierte Datenbanken zu durchsuchen kann der Grover-Algorithmus verwendet werden, welcher einen markierten Zustand in $\mathcal{O}(n) = \sqrt{n}$ finden kann. Weiter kann der Algorithmus auch unter gewissen Umständen für die Lösung von **NP**-Problemen verwendet werden. So verwendet der beste aktuell bekannte Algorithmus zur Lösung von 3-SAT Grovers Algorithmus [16].

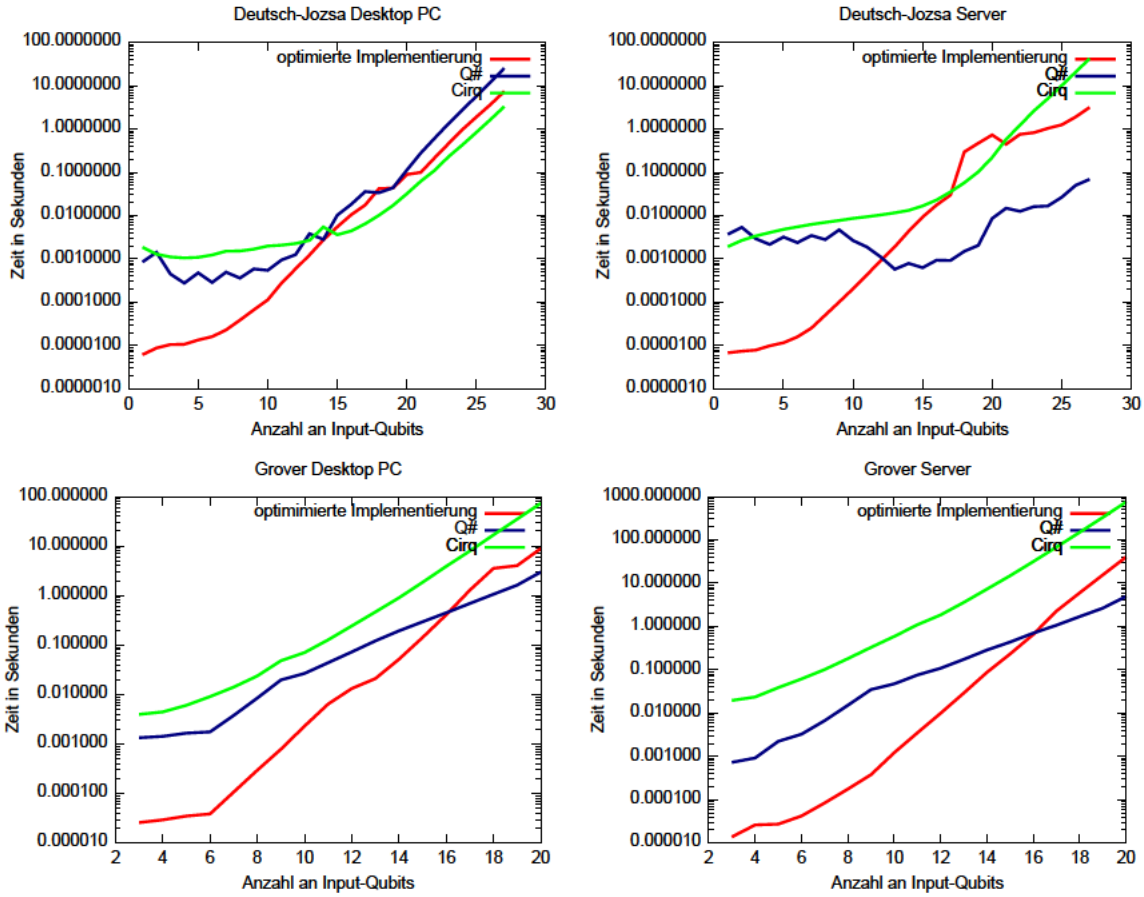

Abbildung 20: Ausführung des Deutsch-Jozsa- und des Grover-Algorithmus auf einem Desktop-PC und einem Server.

Besonders auffällig ist *Circs* gute Performance auf dem Desktop-PC, wohingegen es auf dem Server für fast alle getesteten Eingaben am langsamsten war. Grund dafür ist, dass *Cirq* als einziger Emulator nur auf einem Thread arbeitet, was sich auf CPUs mit geringer Single-Core-Performance sehr negativ auswirkt. Auch wenn *Q#* auf dem Server für stets sehr Performance hat, ist auffällig, dass sich die Laufzeit nicht ansatzweise exponentiell zur Anzahl der Qubits verhält. Die schlechtere Performance auf dem Server für wenige Qubits kann dadurch ausgelöst werden, dass die *.NET*-Umgebung, welche das *Q#*-Programm ausführt, beim Parallelisieren geringer Datenmengen ineffizient sein kann. Dies ist auch auf dem Desktop-PC sichtbar, auf dem Server ist dieser Effekt jedoch wesentlich extremer. Der Grund, weshalb *Q#* auf dem Server bei Ausführung des Deutsch-Jozsa-Algorithmus bei mehr als acht Qubits schneller wurde, konnte nicht gefunden werden. Die Effekte und Optimierungen des Just-in-time Compilers konnten jedoch als Ursache ausgeschlossen werden.

7 Schlussbetrachtung

Insgesamt wurde das Ziel, einen performanten Quantencomputer-Emulator zu entwickeln, erreicht. So ist das finale Produkt unter gewissen Umständen schneller als die Produkte von Google und Microsoft. Weiter erlaubt die entwickelte Implementierung die maximale, von der Hardware her mögliche, Anzahl an verschränkten Qubits zu emulieren. Alle konventionellen Operationen auf Qubits werden vom Emulator unterstützt, sind jedoch teilweise nicht so effizient implementiert, wie bei anderen Emulatoren. So ist die Messung in einer anderen als der Pauli-Z-Basis nicht direkt möglich, stattdessen muss ein entsprechendes Pauli-Gate auf das System angewendet werden, bevor in der Pauli-Z-Basis gemessen wird. Geringe Laufzeiten können mit der entwickelten Implementierung nur erreicht werden, wenn der Quantenalgorithmus effizient implementiert wird. Unnötige Qubits und Gates haben stets einen Einfluss auf die Laufzeit. *Q#* und *Cirq* hingegen sind teilweise in der Lage, schlecht implementierte Algorithmen zu optimieren. So optimiert *Q#* unnötige Gates und Cirq kann nicht verschränkte Qubits mit nur minimaler Laufzeit-Erhöhung mit-emulieren. Diese Unterscheidung von verschränkten und nicht-verschränkten Qubits ermöglicht es *Cirq*, in bestimmten Situationen mit wesentlich geringer Laufzeit zu operieren. Sind beispielsweise alle n Qubits nicht verschränkt, operiert *Cirq* nur auf einem Zustand, während die entwickelte Implementierung auf 2^n Zuständen operiert. Die Verwendung von AVX2 hat den Nachteil, dass das die entwickelte Implementierung nicht auf älterer Hardware ausführbar ist. Da für jede Komponente stets Funktionen ohne SIMD mit fast so guter Laufzeit implementiert wurden, ist die Entwicklung einer Implementierung ohne zwanghafter Verwendung von AVX2 leicht möglich. Das verwenden von double's zur Repräsentierung von Wahrscheinlichkeiten wirkt sich negativ sowohl auf den Speicherverbrauch als auch die Laufzeit aus. So können die meisten AVX2-Befehle mit der selben Anzahl an Befehlen auf doppelt so vielen float 's operieren, als auf double's. Auch wenn ein Umschreiben der finalen Implementierung für die Verwendung von float 's möglich ist, können die durchgeführten Benchmarks keine Informationen darüber geben, inwieweit das Ergebnis optimiert ist. Jedoch sind sowohl *Q#* als auch *Cirq* ebenfalls auf nur einen Typen von Gleitkommazahl festgelegt. So verwendet *Q#* ebenfalls double's, während *Cirq* float 's verwendet.
Der Quellcode der optimierten Implementierung ist unter `https://github.com/Doge815/quantum` verfügbar.

Literatur

[1] A.B. de Avila, R.H.S. Reiser, M.L. Pilla, A.C. Yamin. State-of-the-art quantum computing simulators: Features, optimizations, and improvements for d-gm. 2012.

[2] J. Alman and V. V. Williams. A refined laser method and faster matrix multiplication. 2020.

[3] S. Cao, B. Wu, F. Chen, M. Gong, Y. Wu, Y. Ye, C. Zha, H. Qian, C. Ying, S. Guo, Q. Zhu, H.-L. Huang, Y. Zhao, S. Li, S. Wang, J. Yu, D. Fan, D. Wu, H. Su, H. Deng, H. Rong, Y. Li, K. Zhang, T.-H. Chung, F. Liang, J. Lin, Y. Xu, L. Sun, C. Guo, N. Li, Y.-H. Huo, C.-Z. Peng, C.-Y. Lu, X. Yuan, X. Zhu, and J.-W. Pan. Generation of genuine entanglement up to 51 superconducting qubits. 2023.

[4] R. Duan, H. Wu, and R. Zhou. Faster matrix multiplication via asymmetric hashing. 2022.

[5] R. Gonçalves, M. Amaris†, T. Okada, P. Bruel, and A. Goldman. OpenMP is not as easy as it appears. 2008.

[6] P. I. Hagouel and I. G. Karafyllidis. Quantum computers: registers, gates and algorithms. 2012.

[7] Jonathan Bartlett. *Learn to Program with Assembly: Foundational Learning.* Apress, 2021.

[8] B. Juliá-Díaz, J. M. Burdis, and F. Tabakin. Qdensity—a mathematica quantum computer simulation. 2005.

[9] Y. H. Lee, M. Khalil-Hani, and M. N. Marsono. An fpga-based quantum computing emulation framework based on serial-parallel architecture. 2016.

[10] T. Lin. The seperation of $\mathcal{NP}$ and $\mathcal{PSPACE}$. 2024.

[11] R. Möller. Design of a low-level c++ template simd library. 2016.

[12] E. Nikahd, M. Houshmand, Morteza, S. Zamani, and M. Sedighi. One-way quantum computer simulation. 2015.

[13] Steven S. Skiena. *The Algorithm Design Manual.* Springer, 2008.

[14] V. Strassen. Gaussian elimination is not optimal. 1969.

[15] Yosuke Goto and Minoru Fujishima. Efficient quantum computing emulation system with unitary macro-operations. 2007.

[16] Z. Zhang, R. Paredes, B. Sundar, D. Quiroga, A. Kyrillidis, L. Duenas-Osorio, G. Pagano, and K. R. A. Hazzard. Grover-qaoa for 3-sat: Quadratic speedup, fair-sampling, and parameter clustering. 2024.