Entwurf und Implementierung vektorisierter Sortieralgorithmen

Masterarbeit
zur Erlangung des akademischen Grades
Master of Science (M. Sc.)
im Studiengang Informatik

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik

eingereicht von Mark Blacher

Betreuer: Prof. Dr. Joachim Giesen,
Dr. Lars Kühne

Jena, 01.11.2018

Zusammenfassung

Moderne Prozessoren verfügen über einen Satz an Vektorinstruktionen, mit denen Vektoren bestehend aus mehreren Elementen gleichzeitig verarbeitet werden können. Vektorisierte Algorithmen benutzen überwiegend Vektorinstruktionen und können dadurch eine Verkürzung der Rechenzeit bewirken. Ziel dieser Arbeit ist es, Sortieralgorithmen zu konzipieren, welche mit Vektorinstruktionen implementiert werden können.

Im Einzelnen werden Algorithmen, Vorgehensweisen und Optimierungen vorgestellt, mit denen eine möglichst effiziente Vektorisierung von Sortiernetzwerken, Mergesort und Quicksort möglich ist. Die Implementierung der vektorisierten Sortieralgorithmen erfolgt für den Datentyp `int` mit intrinsischen Funktionen, die den AVX2 Befehlssatz verwenden.

Der performanteste der entwickelten vektorisierten Sortieralgorithmen ist ein Hybrid aus Quicksort und Sortiernetzwerken. Dieser Algorithmus benötigt keinen linear wachsenden zusätzlichen Speicher und hat keinen quadratischen Worst-Case. Der Beschleunigungsfaktor gegenüber `std::sort` liegt zwischen 11 und 12. Im Vergleich zu IPP-Radixsort ist der vektorisierte Sortieralgorithmus aus Quicksort und Sortiernetzwerken mindestens doppelt so schnell.[1]

Die Vektorisierung von Sortieralgorithmen ist auf gegenwärtigen Prozessoren bereits lohnenswert und könnte zur Beschleunigung von sortierintensiven Anwendungen beitragen.

[1] Der Quelltext der Algorithmen und Zeitmessungen kann unter dem folgenden Link heruntergeladen werden: `https://github.com/mark-blacher/masterthesis`.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

1 Einleitung

Sortieren ist ein klassisches Problem in der Informatik. Sortieralgorithmen sind Bestandteil zahlreicher Anwendungen sowohl im kommerziellen als auch im wissenschaftlichen Umfeld. Unter anderen wird Sortierung in der kombinatorischen Optimierung, Astrophysik, Molekulardynamik, Linguistik, Genomik und Wettervorhersage benötigt (siehe Sedgewick 2014, S. 266). Sortierung wird in Datenbanken zur Erstellung von Indices, in Statistik-Software zur Berechnung von Verteilungen oder in der Objekterkennung zur Ermittlung von konvexen Hüllen verwendet. Sortierung ermöglicht die binäre Suche, vereinfacht Probleme wie Test auf Einzigartigkeit von Elementen oder der Bestimmung nächstgelegener Punkte in der Ebene (*closest-pair problem*). Schnellere Sortieralgorithmen können somit die Rechenzeit von Software in den unterschiedlichsten Anwendungsbereichen verkürzen.

Moderne Prozessoren stellen einen Satz an Vektorinstruktionen bereit. Mit einer Vektorinstruktion können mehrere Datenelemente simultan auf Hardwareebene verarbeiten werden, was als Parallelität auf Datenebene bezeichnet wird. Die Größe der Vektorregister beträgt gegenwärtig im Hochleistungsrechnen-Bereich 512 Bit, während Mainstream-CPUs eine Registerbreite von höchstens 256 Bit besitzen. CPUs, die den AVX2 Instruktionssatz unterstützen, sind seit 2013 De-facto-Standard in neuen Laptops und Desktop-PCs. Mit einer Instruktion können in AVX2 256-Bit-Vektoren prozessiert werden. In einen 256-Bit-Vektor passen beispielsweise acht 32-Bit-Integer.

Das Anliegen dieser Arbeit ist die Vektorisierung von Sortieralgorithmen. Im Vordergrund stehen sowohl der Entwurf als auch die Implementierung der vektorisierten Sortieralgorithmen in AVX2. Der algorithmische Schwerpunkt liegt auf Sortiernetzwerken und auf Quicksort, da in Kombination diese Algorithmen den höchsten Speedup gegenüber der skalaren (nichtvektorisierten) Sortierung erzeugen und keinen linear wachsenden Speicher benötigen. Der Vektorisierung von Mergesort ist ebenfalls ein kurzes Kapitel gewidmet, da auch dieser Algorithmus zur effizienten Ausnutzung der Vektorregister geeignet ist.

Damit ein möglichst hoher Speedup gegenüber der skalaren Sortierung entsteht, werden ergänzende Optimierungen der vektorisierten Sortieralgorithmen beschrieben und im Quellcode des Anhangs verwirklicht. Zur Vereinfachung der Darstellung und zur Gewährleistung eines konsistenten Vergleichs der Zeitmessungen erfolgt die Im-

plementierung sämtlicher in der Arbeit vorgestellter vektorisierter Sortieralgorithmen mit intrinsischen Funktionen für den Datentyp `int`. Die Algorithmen sind jedoch universell und auf andere Datentypen bzw. Instruktionssätze übertragbar.

Der wesentliche Inhalt der einzelnen Kapitel der Arbeit ist wie folgt: Nach einer Einführung in die Vektorisierung und den AVX2 Befehlssatz im Kapitel 2 werden im Kapitel 3 verwandte Arbeiten zur vektorisierten Sortierung chronologisch unter Berücksichtigung gegenseitiger Einflüsse der Autoren dargelegt. Das Kapitel 4 widmet sich der Vektorisierung von Sortiernetzwerken. Entsprechend einer Systematisierung der Vektorisierungsstrategien nach schritt- und modulbasiert, werden Vorgehensweisen bzw. Algorithmen zur Vektorisierung von Sortiernetzwerken vorgestellt. In Kapitel 5 findet die Vektorisierung von Mergesort statt. Es wird gezeigt, wie mit Unterstützung der vektorisierten Sortiernetzwerke die skalare Mergeoperation vektorisiert werden kann. Im Mittelpunkt von Kapitel 6 steht die Erstellung eines möglichst schnellen, skalierbaren und für vielseitige Anordnungen der Werte innerhalb des Arrays robusten, vektorisierten Hybridalgorithmus ohne linear wachsenden zusätzlichen Speicher. Hierfür wird eine vektorisierte Variante von Quicksort entwickelt, welche die quadratische Zeitkomplexität im Worst-Case vermeidet und Teilarrays mit gleichen Elementen erkennt. Ferner wird gezeigt, wie eine vektorisierte Pivot-Berechnung mittels eines vektorisierten Zufallszahlengenerators, Gatherbefehlen und eines Median-Netzwerks realisiert werden kann. Eine weiterführende Diskussion zu Themen wie Sortierung von beliebigen einfachen Datentypen bzw. Objekten ist Gegenstand von Kapitel 7.

2 Vektorisierung mit AVX2 Befehlssatz

2.1 Vektorinstruktionen als Bausteine für Vektorisierung

Vektorisierung im Sinne dieser Arbeit bedeutet einen Algorithmus derart zu transformieren, dass dieser Vektorinstruktionen verwendet. Moderne Prozessoren stellen einen Satz an Vektorinstruktionen zur parallelen Verarbeitung der Datenelemente eines Vektorregisters bereit. Die Anzahl der Elemente, die gleichzeitig verarbeitet werden können, ist abhängig von der Registerbreite und den Datentyp der Elemente. Bei einer Registerbreite von 256 Bit können simultan beispielsweise acht 32-Bit-Integer mit einer Vektorinstruktion manipuliert werden. Diese Form der Parallelisierung auf Datenebene entspricht dem SIMD (*Single Instruction, Multiple Data*) Paradigma zur Klassifizierung von parallelen Rechnern von Flynn (1972). Mit einer Instruktion werden mehrere Datenflüsse ausgelöst, auf denen dieselbe Operation zeitgleich angewendet wird.[1]

Vektorerweiterungen im x86-Instruktionssatz existieren seit 1997 (siehe. Bryant 2016, S. 294). Die Registerbreite steigt seitdem beständig. Angefangen mit 64 Bit bei MMX (*Multi Media Extension*) stieg die Registerbreite auf 128 Bit bei SSE (*Streaming SIMD Extensions*) und verdoppelte sich erneut auf 256 Bit bei AVX (*Advanced Vector Extensions*). Gegenwärtige Intel CPUs im Hochleistungsrechnen-Bereich haben eine Registerbreite von 512 Bit und unterstützen den Befehlssatz AVX-512 (vgl. Intel Corporation 2016b). Moderne Mainstream-CPUs hingegen unterstützen höchstens AVX2. Wie sein Vorgänger AVX verwendet AVX2 Vektoren von maximal 256 Bit. Alle in den folgenden Kapiteln vorgestellten vektorisierten Sortieralgorithmen sind mit AVX bzw. AVX2 Instruktionen implementiert.

Mit jeder Erweiterung des Instruktionssatzes ergeben sich für Algorithmen neue Vektorisierungsmöglichkeiten. Beispielsweise ist es durch die in AVX2 eingeführten Gatherbefehle möglich, die Berechnung des Pivot-Werts in Quicksort vollständig zu

[1] Ein anderes Paradigma zur Klassifizierung von parallelen Rechnern ist MIMD (*Multiple Instruction, Multiple Data*), das die parallele Bearbeitung von verschiedenen Daten mit jeweils verschiedenen Befehlen beschreibt (vgl. Flynn 1972; siehe Herold 2017, S. 288). Moderne Mehrprozessorsysteme bzw. Mehrkernprozessoren realisieren MIMD.

vektorisieren. Ebenfalls existieren seit AVX2 Permutations-Befehle, die das Vertauschen von acht 32-Bit-Elementen in beliebiger Reihenfolge innerhalb des Vektorregisters erlauben, was nützlich zur Vektorisierung von allen im Rahmen dieser Arbeit dargestellten Sortieralgorithmen ist.

2.2 Latenz und Durchsatz von Instruktionen

Latenz und Durchsatz sind zwei Kennzahlen, die Auskunft über die Performanz einer Instruktion geben. Latenz ist die Anzahl der Takte, die eine Instruktion benötigt, bis ihr Ergebnis einer anderen Instruktion zur Verfügung steht. Eine Instruktion mit einer Latenz von vier braucht, sobald sie startet, vier Takte, bis ihr Resultat von einer anderen Instruktion verwendet werden kann. Der Durchsatz gibt dagegen an, wie viele Takte die Ausführung der Instruktion dauert. Eine Instruktion mit einem Durchsatz von zwei beansprucht das Rechenwerk, worauf sie ausgeführt wird, für zwei Taktzyklen. Instruktionen, die zur Ausführung dasselbe Rechenwerk benötigen, müssen zwei Taktzyklen warten. Die Latenz ist immer größer-gleich dem Durchsatz, da diese die Ausführungszeit einer Instruktion implizit beinhaltet (siehe Developer Zone 2008).

Die Latenz einer Instruktion wird von der Komplexität des zugrundeliegenden Rechenwerks beeinflusst. Ein Shuffle, bei dem die Elemente innerhalb der 128-Bit-*Lanes* vertauscht werden, benötigt ein kleineres weniger komplexes Rechenwerk als eine Permutation, welche die Elemente zwischen den 128-Bit-*Lanes* vertauscht. Dementsprechend hat der Shuffle eine Latenz von einem Takt und die Permutation eine Latenz von drei Takten, obwohl beide Instruktionen 256-Bit-Register in AVX bzw. AVX2 benutzen (vgl. Intrinsics Guide 2018). Reicht eine Vertauschung der Elemente innerhalb der 128-Bit-*Lanes* aus, dann ist der Shuffle die effizientere Instruktion im Vergleich zur Permutation. Es ist anzunehmen, dass die Latenz der Permutation mit zunehmender Registerbreite steigen wird, da die benötigte Hardwarefläche zur Implementierung eines Vertauschungsnetzwerks proportional zum Quadrat der Eingabegröße ist (siehe Chhugani et al. 2008, S. 1315). Vektor-Instruktion wie Minimum oder Maximum, welche unabhängig die Elemente des Vektors verarbeiten, leiden dagegen nicht an steigenden Latenzen mit zunehmender Registerbreite.

Ähnlich wie beim Pipelining, bei dem die verschiedenen Phasen der Befehlsbearbeitung parallelisiert werden (vgl. Herold 2017, S. 643), können Vektorinstruktionen mit langer Latenz und kurzem Durchsatz, die ein gemeinsames Rechenwerk nutzen, der Reihe nach ohne Verzögerung ausgeführt werden. Damit dies möglich ist, müssen die Instruktionen eng beieinander liegen und voneinander unabhängig sein.

2.3 Intrinsische Funktionen für AVX2

Statt den vektorisierten Teil der Algorithmen mit Assemblerbefehlen zu verfassen, werden intrinsische Funktionen benutzt. Intrinsische Funktionen haben den Vorteil, dass explizite Zuweisungen der Register entfallen. Mit intrinsischen Funktionen entscheidet der Compiler, in welchen Vektorregistern die Berechnungen stattfinden. Eine intrinsische Funktion kann mehrere Instruktionen zusammenfassen. Im Folgenden werden die für Vektorisierung der Sortierung von 32-Bit-Integern wesentlichsten intrinsischen Funktionen erläutert. Bei der Vorstellung der Sortieralgorithmen kommen weitere Instruktionen hinzu. Es wird nicht zwischen AVX und AVX2 unterschieden, da jede CPU, die AVX2 unterstützt, auch AVX Instruktionen ausführen kann.

Ein 256-Bit-Integer-Vektor ist von Datentyp `__m256i`. Er bietet Platz für 32 `char`, 16 `short`, 8 `int` oder 4 `long` Ganzzahlen. Diese Werte können vorzeichenbehaftet oder vorzeichenlos sein. Bevor mit Vektoren gerechnet werden kann, muss man diese mit Daten initialisieren. Einerseits ist es möglich, die Daten aus dem Speicher zu laden, andererseits kann ein Vektor direkt mit skalaren Werten initialisiert werden (vgl. Quelltext 2.1).

Quelltext 2.1: Initialisierung von Vektoren

```
1 int arr[8] = {3, 4, 15, 6, 16, 10, 5, 8};
2 /* lade Daten von einer nicht ausgerichteten Speicheradresse */
3 __m256i v_loaded = _mm256_loadu_si256(reinterpret_cast<__m256i *>(arr));
4 /* erstelle Vektor mit acht unterschiedlichen Integern */
5 __m256i v_different = _mm256_setr_epi32(1, 2, 3, 4, 5, 6, 7, 8);
6 /* erstelle Vektor mit acht gleichen Integern */
7 __m256i v_same = _mm256_set1_epi32(7);
```

Die Vektoren, die als Parameter in eine intrinsische Funktion eingehen, sind nach Ausführung der intrinsischen Funktion unverändert. Damit sich Vektoren nach der Berechnung verändern, müssen sie überschrieben werden. Der Quelltext 2.2 zeigt wie die Permutation bzw. der Shuffle realisiert werden.

Quelltext 2.2: Permutation und Shuffle

```
1 __m256i indexes = _mm256_setr_epi32(0, 1, 6, 3, 7, 5, 2, 4);
2 /* v_loaded permutiert von <3 4 15 6 16 10 5 8> zu <3 4 5 6 8 10 15 16> */
3 v_loaded = _mm256_permutevar8x32_epi32(v_loaded, indexes);
4 /* v_different shuffelt von <1 2 3 4 5 6 7 8> zu <4 3 2 1 8 7 6 5> */
5 v_different = _mm256_shuffle_epi32(v_different, 0b00011011);
```

Der Vektor `indexes` enthält die Indices, nach denen `v_loaded` permutiert wird. Im Beispiel ist `v_loaded` nach der Umstellung seiner Werte sortiert. Der Shuffle (`_mm256_shuffle_epi32`) benötigt im zweiten Parameter Indices, die angeben, wie die Werte innerhalb der zwei 128-Bit-*Lanes* symmetrisch zu vertauschen sind. Im

Unterschied zur Permutation, bei welcher die Indices zur Laufzeit bestimmt werden können, müssen die Indices des Shuffles eine 8-Bit-Kompilierzeitkonstante sein. Die 8-Bit-Kompilierzeitkonstante ist von rechts nach links, jeweils zwei-bitweise zu lesen, d. h. `00 01 10 11` sind von rechts gelesen die Indices `4 3 2 1`. Dementsprechend kehrt die Kompilierzeitkonstante `0b00011011` im Shuffle die Reihenfolge der Werte innerhalb der 128-Bit-*Lanes* um.

Zum Vermischen von zwei Vektoren dient die Funktion `_mm256_blend_epi32`. Diese hat drei Parameter. Die ersten beiden Parameter sind für die zwei Vektoren bestimmt die vermischt werden. Der dritte Parameter ist eine 8-Bit-Kompilierzeitkonstante und gibt an, wie die beiden Vektoren zu vermischen sind. Die Kompilierzeitkonstante ist von rechts nach links bitweise zu lesen, wobei `0` bedeutet, dass der Wert des ersten Vektors und `1` der des zweiten Vektors gewählt wird (vgl. Quelltext 2.3).

Quelltext 2.3: Vermischung von zwei Vektoren

```
1 __m256i v1 = _mm256_setr_epi32(1, 10, 11, 4, 5, 14, 7, 16);
2 __m256i v2 = _mm256_setr_epi32(9, 2, 3, 12, 13, 6, 15, 8);
3 /* v_blended ist <1 2 3 4 5 6 7 8> */
4 __m256i v_blended = _mm256_blend_epi32(v1, v2, 0b10100110);
```

Für die Berechnung des paarweisen Minimums bzw. Maximums zwischen den Elementen von zwei Vektoren gibt es die Funktionen `_mm256_min_epi32` bzw. `_mm256_max_epi32`. Im Quelltext 2.4 wird aus den in Quelltext 2.3 definierten Vektoren `v1` und `v2` das paarweise Minimum bzw. Maximum ermittelt.

Quelltext 2.4: Paarweises Minimum bzw. Maximum

```
1 /* v_min ist <1 2 3 4 5 6 7 8> */
2 __m256i v_min = _mm256_min_epi32(v1, v2);
3 /* v_max ist <9 10 11 12 13 14 15 16> */
4 __m256i v_max = _mm256_max_epi32(v1, v2);
```

Damit die angewendeten Operationen auf den Vektor nicht verloren gehen, wird dieser ins Array zurückgespeichert. Hierfür wird die Funktion `_mm256_storeu_si256` benutzt. Der erste Parameter ist ein Zeiger auf den Speicherort, der zweite Parameter ist der zu speichernde Vektor (vgl. Quelltext 2.5).

Quelltext 2.5: Speicherung von Vektoren

```
1 /* v_loaded nach Array arr speichern,
2  * Speicheradresse braucht nicht ausgerichtet zu sein */
3 _mm256_storeu_si256(reinterpret_cast<__m256i *>(arr), v_loaded);
```

Um ein Programm zu erstellen, das intrinsische Funktionen für AVX2 verwendet, muss die Header-Datei `immintrin.h` eingebunden und kompilerspezifische *Flags* gesetzt werden (z. B. `-mavx2` für GCC).

3 Verwandte Arbeiten

3.1 Sortieren auf Vektor-Supercomputern

Siegel (1977) beschreibt, wie eine Vektorisierung von Bitonic Mergesort auf SIMD-Maschinen erfolgen kann.

Stone (1978) vektorisiert Quicksort und eine von ihm selbst adaptierte Variante des Bitonic Mergesort auf dem Vektor-Supercomputer CDC STAR. Zur Implementierung der Partitionierungsfunktion von Quicksort benötigt Stone ein temporäres Array zum Herauskopieren der Werte, die größer-gleich dem Pivot-Wert sind. Bitonic Mergesort wird lediglich für Zweierpotenzen als Arraylänge implementiert. Quicksort und Bitonic Mergesort werden getrennt voneinander betrachtet, ohne eine Synthese dieser in einem Hybridalgorithmus anzustreben.

Levin (1990) konstruiert einen vektorisierten Hybridalgorithmus aus Quicksort und Odd-even Transposition Sort[1] für die Vektor-Supercomputer Cray X-MP und Convex. Die Partitionierungsfunktion von Quicksort entspricht im Wesentlichen der von Stone (1978).[2] Der vektorisierte Odd-even Transposition Sort wird für die Sortierung kurzer Subpartitionen verwendet. Beansprucht die Subpartition weniger als die Hälfte des Vektorregisters, dann stellt Levin fest, dass der skalare Insertion Sort schneller im Vergleich zum vektorisierten Odd-even Transposition Sort ist. Levin schlägt die Verwendung von effizienteren Sortiernetzwerken als Odd-even Transposition Sort zur Sortierung kurzer Subpartitionen vor.

Zagha und Blelloch (1991) vektorisieren Radixsort auf dem Vektor-Supercomputer CRAY Y-MP. Im Vergleich zum Hybridalgorithmus aus Quicksort und Odd-even Transposition Sort von Levin (1990) erreichen Zagha und Blelloch mit ihren vektorisierten Radixsort eine Beschleunigung um den Faktor drei bis fünf. Der Speedup ist

[1] Odd-even Transposition Sort ist ein dem Bubblesort verwandtes Sortiernetzwerk. Zur Sortierung eines Arrays mit n Elementen, benötigt der Algorithmus n parallele Schritte (vgl. Nielsen 2016, S. 113). Eine skalare Implementierung des Odd-even Transposition Sort hat wie Bubblesort eine Komplexität von $\mathcal{O}(n^2)$. Ein namensverwandter Sortiernetzwerkalgorithmus ist der Odd-even Mergesort. Odd-even Mergesort hat aber eine Komplexität von $\mathcal{O}(n \log^2 n)$ und ist nicht zu verwechseln mit dem weniger effizienten Odd-even Transposition Sort.

[2] Sowohl Stone (1978) als auch Levin (1990) verwenden vektorisierte Komprimierungsbefehle zur Realisierung der Partitionierungsfunktion von Quicksort. Erst AVX-512 stellt ähnliche Komprimierungs-Instruktionen (z. B. `_mm512_mask_compress_epi32`) bereit (siehe. Intrinsics Guide 2018).

zum Teil darauf zurückzuführen, dass CRAY Y-MP keinen Cache hat, d. h. der Zugriff auf alle Datenelemente des Arbeitsspeichers erfolgt mit denselben Zeitaufwand.[3] Die Performanz des vektorisierten Radixsort hängt insbesondere von der Performanz der Gather und Scatter Operationen ab.[4] Auf Maschinen ohne Cache haben Gather und Scatter Instruktionen im Vergleich zu Maschinen mit Cache geringere Kosten.

3.2 Sortieren auf modernen Architekturen

Inoue, Moriyama et al. (2007) entwerfen einen zweiphasigen vektorisierten parallelen Sortieralgorithmus. Zur Implementierung benutzen Inoue, Moriyama et al. 128-Bit VMX (*Vector Media Extensions*) Instruktionen von IBM. In der ersten Phase wird das Array in Blöcke aufgeteilt, die jeweils in den L2 Cache eines Prozessorkerns passen. Jeder Block wird mit einer vektorisierten Variante von Combsort sortiert. Das Array muss hierfür an den 128-Bit-Grenzen ausgerichtet im Speicher vorliegen und die Anzahl der Elemente darin ein Vielfaches von vier sein. Die Durchschnittskomplexität der ersten Phase beträgt $\mathcal{O}(n \log n)$. In der zweiten Phase werden die sortierten Blöcke solange gemergt, bis das komplette Array sortiert ist. Der vektorisierte Mergealgorithmus, den Inoue, Moriyama et al. entwickeln, ist ein Hybrid aus Odd-Even Merge und der gewöhnlichen Mergeoperation bei Mergesort. Die Komplexität für eine einzelne Mergeoperation in der zweiten Phase beträgt $\mathcal{O}(n)$ und ist damit geringer als die Komplexität $\mathcal{O}(n \log n)$ von Odd-Even Merge oder Bitonic Merge.[5] Die sequentielle Variante des zweiphasigen Algorithmus erzielt eine Beschleunigung von Faktor drei bei der Sortierung von 32-Bit-Integern ggü. der GNU C++ STL.

Chhugani et al. (2008) produzieren ebenfalls wie Inoue, Moriyama et al. (2007) einen zweiphasigen vektorisierten cache-optimierten parallelen Sortieralgorithmus. In der ersten Phase wird das Array in Blöcke aufgeteilt und jeder Block sortiert. Auf der untersten Ebene verwenden Chhugani et al. einen vektorisierten Odd-even Mergesort. Die Elemente werden hierfür vertikal sortiert und anschließend transponiert. Zum Mergen der einzelnen sortierten Gruppen wird ein vektorisierter Hybrid aus Bitonic Mergesort und der gewöhnlichen Mergeoperation bei Mergesort implementiert. In der zweiten Phase erfolgt der Merge der einzelnen Blöcke mit einer ähnlichen Merging-Strategie, wobei für besonders lange Arrays Multiway Merg zum Einsatz kommt.

[3] Zur Vermeidung langsamer Zugriffe auf den Arbeitsspeicher besitzen moderne Rechnerarchitekturen mehrere Cache-Ebenen mit unterschiedlicher Nähe zur CPU (siehe Herold 2017, S. 653 ff.).

[4] Scatter Instruktionen sind in AVX-512 verfügbar, jedoch nicht in AVX2 (siehe. Intrinsics Guide 2018).

[5] Odd-Even Merge bzw. Bitonic Merge ist der Teil des Sortiernetzwerks von Odd-Even Mergesort bzw. Bitonic Mergesort, der für den Merge zwischen zwei sortierten Arraypartitionen verantwortlich ist.

Mittels eines simulierten Prozessors evaluieren Hayes et al. (2015) vektorisierte Varianten der Sortieralgorithmen Quicksort, Bitonic Mergesort und Radixsort hinsichtlich ihres Zukunftspotentials.[6] In der experimentellen Konfiguration ist die Breite der Register veränderbar. Ferner erstellen Hayes et al. vektorisierte Instruktionen für den simulierten Prozessor, die für eine effiziente Implementierung der Algorithmen notwendig sind.[7] Zur Bewertung werden drei unterschiedlich lange Arrays mit gleichverteilten Zufallszahlen benutzt. Die Vektorisierung von Quicksort erfolgt wie bei Levin (1990) mit Odd-even Transposition Sort zur Sortierung kurzer Subpartitionen. Als Pivot-Strategie dient Median-aus-drei. Den Bitonic Mergesort vektorisieren Hayes et al. indem sie zuerst Blöcke von doppelter Registerbreite sortieren und anschließend solange mergen, bis das komplette Array sortiert ist. Für die Vektorisierung von Radixsort wird der Algorithmus von Zagha und Blelloch (1991) benutzt. Die Autoren schlussfolgern, dass die Vektorisierung von Radixsort das größte Zukunftspotential hat, wenn der fragmentierte Speicherzugriff und Replikationen interner Strukturen behoben werden.[8] Aufbauend auf Radixsort entwerfen Hayes et al. einen cache-freundlichen, vektorisierten Sortieralgorithmus mit effizienten unit-stride Speicherzugriff und ohne replizierte interne Strukturen. Zur Implementierung des Algorithmus werden zwei neue Vektor-Instruktionen eingeführt und ihre mögliche Hardwarerealisierung dargestellt. Hayes et al. messen eine durchschnittliche Verbesserung der Performanz um Faktor 3,4 ihres Algorithmus zum nächstbesten der drei untersuchten Sortieralgorithmen.

Gueron und Krasnov (2016) vektorisieren die Partitionierungsfunktion von Quicksort auf den AVX bzw. AVX2 Befehlssatz. Ihre Partitionierungsfunktion benötigt zusätzlichen Speicher. Der Pivot-Wert ist immer das letzte Element der Partition, sodass der vektorisierte Quicksort stabil ist. Nach der vektorisierten Vergleichsoperation mit dem Pivot-Wert werden die Elemente im Vektorregister mithilfe von in einer Lookup-Tabelle gespeicherten Shuffle-Masken komprimiert.[9] Sobald eine Subpartition weniger als 32 Elemente enthält, erfolgt ähnlich wie bei der STL ein Wechsel auf den skalaren Insertionsort. Getestet wird der Algorithmus für verschiedene Datentypen auf unterschiedlich langen mit gleichverteilten Zufallszahlen befühlten Arrays. Die

[6] Hayes et al. verwenden laut Paper eine vektorisierte *in-place* Variante von Quicksort, d. h ohne zusätzlichen Speicher. Die Partitionierungsfunktion der *in-place* Variante ist jedoch nicht im Paper enthalten.

[7] Die meisten der simulierten Instruktionen sind ebenfalls in AVX-512 vorhanden.

[8] Beim vektorisierten Radixsort von Zagha und Blelloch (1991) kommt es zu Wiederholungen interner Strukturen, deren Größe von Radix und die Anzahl von der Registerbreite abhängt. Dies stellt sich als Flaschenhals heraus, falls eine Beschleunigung mit größeren Radix bzw. breiteren Registern beabsichtigt wird.

[9] Gueron und Krasnov merken an, dass eine Lookup-Tabelle in AVX-512 unnötig ist und durch Komprimierungsbefehle ersetzt werden kann.

Ausführungszeiten des vektorisierten Quicksort bei der Sortierung von 32-Bit-Integern liegen oberhalb derjenigen von IPP-Radixsort[10].

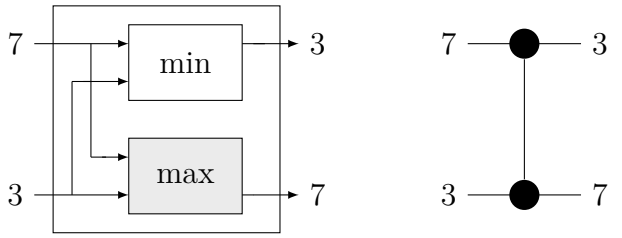
Bramas (2017) erstellt einen vektorisierten Hybridalgorithmus aus Quicksort und Bitonic Mergesort. Zur Implementierung wird der Befehlssatz AVX-512 benutzt. Mit der Verwendung neuer Komprimierungsbefehle in AVX-512 implementiert Bramas die Partitionierungsfunktion von Quicksort ohne die Zuhilfenahme von vorausberechneten Permutations- bzw. Shuffle-Masken. Durch das Zwischenspeichern der beiden an den äußeren Enden des Arrays liegenden SIMD-Vektoren gelingt es Bramas die Partitionierungsfunktion ohne zusätzlichen Speicher zu realisieren. Der Pivot-Wert zur Partitionierung des Arrays wird mit Median-aus-drei bestimmt. Die Partitionierungsfunktion wird in der ersten Phase des Algorithmus verwendet, bis die Arraylänge für 32-Bit-Integer kleiner-gleich 256 ist. In der zweiten Phase des Algorithmus werden die übriggeblieben Elemente mit einem verzweigungsfreien Bitonic Mergesort sortiert. Bitonic Mergesort wird hierbei für unterschiedliche Arraylängen auskodiert und nicht als kompakte algorithmische Berechnungsvorschrift formuliert. Für das Sortieren von gleichverteilten Zufallszahlen in langen Arrays erreicht der Hybridalgorithmus einen Speedup von 1,4 im Vergleich zum IPP-Radixsort.

[10] IPP-Radixsort ist enthalten in der Softwarebibliothek *Integrated Performance Primitives* von Intel.

4 Sortiernetzwerke

4.1 Vergleichsmodule und Sortiernetzwerke

Vergleichsmodule sind die Bausteine der Sortiernetzwerke. Ein Vergleichsmodul besteht aus zwei Eingängen und zwei Ausgängen. Jeder Eingang empfängt eine Zahl. Die zwei Zahlen werden im Vergleichsmodul sortiert und an die beiden Ausgänge übergeben, wobei ein Ausgang stets die kleinere und der andere die größere Zahl herausgibt (siehe Vöcking et al. 2008, S. 32 ff.). In folgenden Ausführungen wird die kleinere Zahl immer an den oberen und die größere Zahl an den unteren Ausgang geleitet. Die Abb. 4.1 zeigt zwei Darstellungsmöglichkeiten für Vergleichsmodule. Zur Zeichnung von Sortiernetzwerken wird überwiegend die vereinfachte Variante von Abb. 4.1b verwendet.



(a) Detailliert **(b)** Vereinfacht

Abb. 4.1: Darstellungsmöglichkeiten für Vergleichsmodule

Vergleichsmodule lassen sich als Hardware realisieren (vgl. Parhami 2002, S. 131 f.) oder in Software implementieren. Aus Vergleichsmodulen können Netzwerke erstellt werden. Die Ausgänge der vorhergehenden Vergleichsmodule sind die Eingänge der nachfolgenden. Netzwerke, die eine feste Anzahl an Elementen immer sortieren, werden als Sortiernetzwerke bezeichnet. Ein Sortiernetzwerk, das eine Zahlenfolge der Länge vier sortiert, ist in Abb. 4.2 auf der nächsten Seite dargestellt. Die Eingabe kommt von links und läuft nach rechts durch das Netzwerk hindurch. Insgesamt besteht dieses Sortiernetzwerk aus fünf Vergleichsmodulen. Die Zahlen über den Vergleichsmodulen stehen für die parallelen Schritte. Jeder parallele Schritt ist dadurch charakterisiert, dass keine Abhängigkeiten zwischen den einzelnen Modulen vorliegen. Alle Vergleichsmodule innerhalb der grauen Box können somit gleichzeitig

ausgeführt werden. Das Netzwerk in Abb. 4.2 hat insgesamt drei parallele Schritte. Die fettgedruckte zwei über der zweiten grauen Box bedeutet, dass eine Merging-Phase beginnt. Im ersten Schritt werden die Zahlen in Zweiergruppen sortiert. Im Schritt zwei und drei erfolgt der Merge[1] der beiden Zweiergruppen.[2]

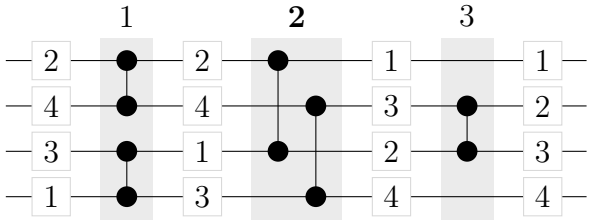


Abb. 4.2: Odd-Even Mergesort für 4 Elemente (5 Module, 3 Schritte)

Selbst wenn ein Sortiernetz eine bereits sortierte Zahlenfolge, als Input erhält, werden alle Vergleichsoperationen ausgeführt. Sortiernetzwerke zeichnen sich somit dadurch aus, dass Vergleiche datenunabhängig durchgeführt werden. Dies ermöglicht ihre Realisierung als Schaltungen in Hardware (siehe Huang et al. 2001). Die Implementierung von Sortiernetzwerken in Software benötigt keinen extra Speicher, alle Vergleichs- und Austauschoperationen können *in-place* vorgenommen werden. Eine nicht-vektorisierte serielle Implementierung in C++ des Sortiernetzwerks aus Abb. 4.2 zeigt Quelltext 4.1.

Quelltext 4.1: Serielles nicht-vektorisiertes Sortiernetzwerk

```
1 /* Makro COEX (compare-exchange) simuliert Vergleichsmodul */
2 #define COEX(a, b) if(a > b) {int c = a; a = b; b = c;}
3 /* Sortiernetzwerk für 4 Integer */
4 inline void sort_4_int(int *arr){
5   COEX(arr[0], arr[1]); COEX(arr[2], arr[3]); /* Schritt 1 */
6   COEX(arr[0], arr[2]); COEX(arr[1], arr[3]); /* Schritt 2 */
7   COEX(arr[1], arr[2]);}                      /* Schritt 3 */
```

Die Anzahl der parallelen Schritten und die Menge an Vergleichsmodulen sind die beiden Kennzahlen, nach denen Sortiernetzwerke optimiert werden. Ein Sortiernetzwerk mit der minimalen Anzahl an Vergleichsmodulen hat nicht notwendig die wenigsten parallelen Schritten und umgekehrt genauso. Eine Übersicht der oberen und unteren Grenzen der Kennzahlen zu den besten bekannten Sortiernetzwerken für n Eingabeelemente und $n \leq 20$ gibt Tabelle 4.1 auf der nächsten Seite. Falls in der zweiten Zeile der jeweiligen Kennzahl ein Wert steht, dann handelt es sich um die untere Grenze. Steht keine Angabe darunter, dann entspricht die obere der

[1] Als Merge wird im Allgemeinen eine Operation bezeichnet, die aus zwei sortierten Zahlenfolgen eine sortierte Zahlenfolge erstellt, wobei alle Elemente der beiden ursprünglichen Zahlenfolgen darin enthalten sind.

[2] Siehe Abb. A.1 auf Seite 73 für ein ausführliches Beispiel zur Sortierung von acht Elementen.

Tab. 4.1: Kennzahlen der besten Sortiernetzwerke für $n \leq 20$

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Module	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60	71	78	86	92
											33	37	41	45	49	53	58	63	68	73
Schritte	0	1	3	3	5	5	6	6	7	7	8	8	9	9	9	9	10	11	11	11
																		10	10	10

unteren Grenze und es ist bewiesen, dass das bekannte Sortiernetzwerk optimal ist. Zum Beispiel für $n = 11$ hat das optimale Sortiernetzwerk zwischen 33 und 35 Vergleichsmodule, wie viele genau, ist unbekannt (vgl. Codish et al. 2016).

4.2 Batchers algorithmische Sortiernetzwerke

Im Jahr 1968 erschien ein Artikel von Batcher, worin zwei Merge-Strategien (Bitonic Merge und Odd-Even Merge) für beliebig lange Zahlenfolgen beschrieben sind. Für den Spezialfall, n ist eine Zweierpotenz, erstellte Batcher mit den Merge-Strategien zwei Verfahren zur Generierung von Sortiernetzwerken.[3] Beide Verfahren mergen zuerst die einzelnen Elemente zu Zweiergruppen. Anschließend werden die Zweiergruppen zu Vierergruppen gemergt. Die Vierergruppen werden zu sortierten Achtergruppen vereinigt usw. Nach dem letzten Merge sind alle Elemente sortiert. Die zwei Algorithmen zur Realisierung dieser Art von Sortiernetzwerken sind Bitonic Mergesort und Odd-Even Mergesort. In Abb. 4.3 auf der nächsten Seite sind Batchers Sortiernetzwerke für 16 Elemente dargestellt.[4]

Beide Sortiernetzwerke benötigen für die Sortierung von 2^p Elementen, $\frac{p(p+1)}{2}$ parallele Schritte[5] und haben bezüglich der Anzahl an Vergleichsmodulen die gleiche asymptotische Komplexität von $\mathcal{O}(n \log^2 n)$. Die tatsächliche Anzahl der Vergleichsmodule ist bei Odd-Even Mergesort mit $(p^2 - p + 4)2^{p-2} - 1$ geringer als bei Bitonic Mergesort mit $(p^2 + p)2^{p-2}$. Für $p \leq 3$, d. h. $n \in \{1, 2, 4, 8\}$ produziert Odd-Even Mergesort sowohl hinsichtlich der parallelen Schritte als auch der Anzahl an Vergleichsmodulen die bestmöglichen Sortiernetzwerke aus Tabelle 4.1.[6]

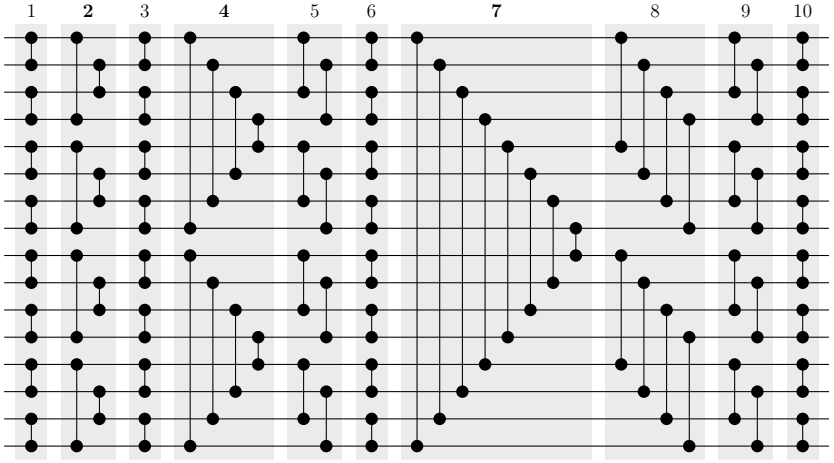
In Software implementiert sind Vergleichsmodule bei Sortiernetzwerken nichts anderes als Vergleichsoperationen bei Sortierverfahren wie Mergesort oder Quick-

[3] Knuth (1998, S. 223–225, 230–232) demonstriert, wie mit Odd-Even Merge und Bitonic Merge algorithmisch beliebig lange Zahlenfolgen sortiert werden können.

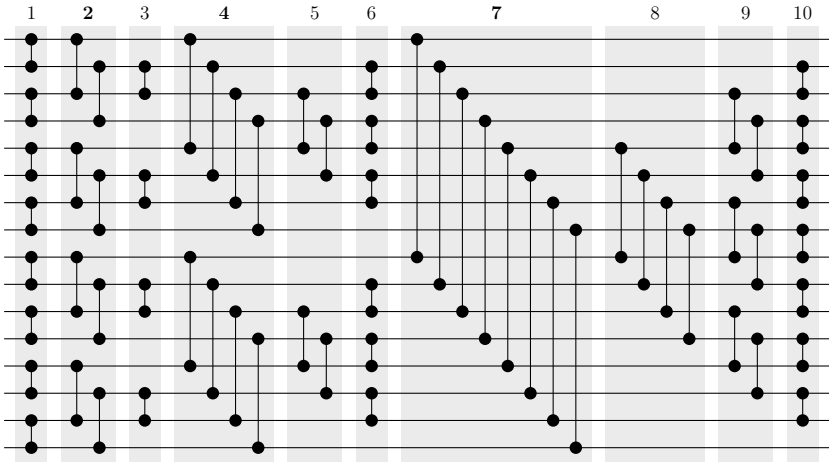
[4] Die Sortiernetzwerke für 32 Elemente befinden sich im Anhang auf den Seiten 73 bis 74.

[5] Für $p = 4$, das bedeutet 16 Elemente, ergeben sich 10 parallele Schritte.

[6] Ein algorithmisches Netzwerk mit gleicher Anzahl an Vergleichsmodulen und parallelen Schritten wie das Odd-Even Sortiernetzwerk ist das Pairwise Sorting Network von Parberry (1992). Für die Selektion von k kleinsten bzw. größten Elemente aus n hat Pairwise Sort dagegen tendenziell weniger Vergleichsmodule als Odd-Even Mergesort (Zazon-Ivry und Codish 2012).



(a) Bitonic Mergesort (80 Module, 10 Schritte)



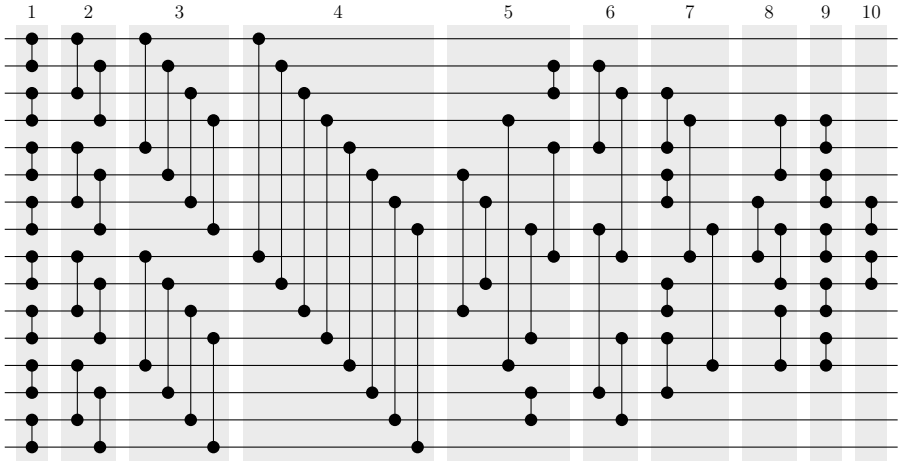
(b) Odd-Even Mergesort (63 Module, 10 Schritte)

Abb. 4.3: Bitonic Mergesort und Odd-Even Mergesort für 16 Elemente

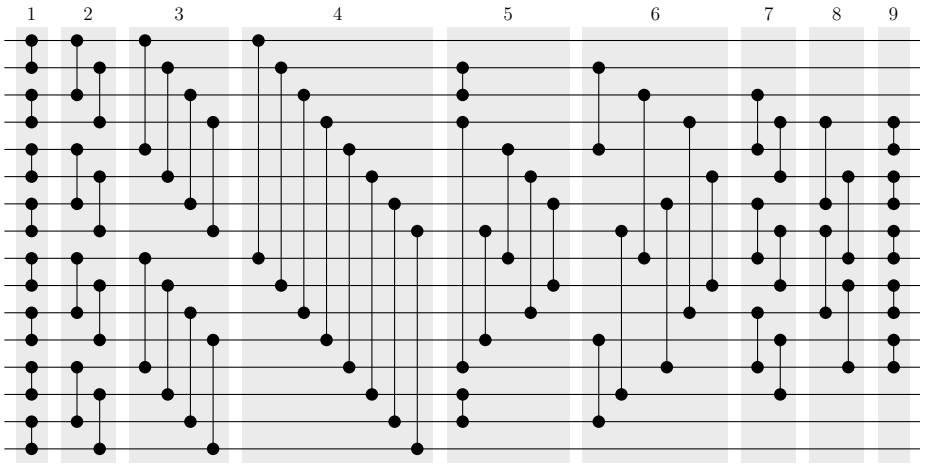
sort. Es ist bewiesen, dass eine auf Vergleichsoperationen basierte Sortierung eine asymptotische Komplexität im Worst-Case von $\mathcal{O}(n \log n)$ nicht unterschreiten kann (siehe für Beweis Cormen et al. 2009, S. 191–193). Mit asymptotisch $\mathcal{O}(n \log^2 n)$ Vergleichsoperationen sind Bitonic Mergesort und Odd-Even Mergesort nicht optimal. Ein einzelner Merge hat bei Batchers Netzwerken eine Komplexität von $\mathcal{O}(n \log n)$. Optimal wäre wie bei Mergesort eine Komplexität von $\mathcal{O}(n)$. In Abb. 4.3 ist die höhere Komplexität der Sortiernetzwerke im Vergleich zu Mergesort unter anderem dadurch erkennbar, dass zwischen den fettgedruckten Zahlen über den grauen Boxen, die für den Beginn einer Mergeoperation stehen, mit jedem neuen Merge ein paralleler Schritt hinzukommt. Theoretisch können Sortiernetzwerke mit einer Komplexität von $\mathcal{O}(n \log n)$ konstruiert werden, diese sind aber aufgrund eines zu hohen konstanten Faktors für die Praxis ungeeignet (vgl. Vöcking et al. 2008, S. 40).

4.3 Beste Sortiernetzwerke für 16 Elemente

Odd-Even Mergesort von Batcher aus Abb. 4.3b auf Seite 26 ist für 16 Elemente hinsichtlich der parallelen Schritte und der Anzahl an Vergleichsoperation suboptimal. In Abb. 4.4 sind die besten bekannten Sortiernetzwerke für 16 Elemente dargestellt. Es ist nicht bekannt, wie die Konstruktion dieser Sortiernetzwerke generalisiert werden kann, um größere Sortiernetzwerke zu erstellen (vgl. Knuth 1998, S. 226 ff.).[7]



(a) 60 Module, 10 Schritte



(b) 61 Module, 9 Schritte

Abb. 4.4: Beste Sortiernetzwerke für 16 Elemente

Algorithmische Sortiernetzwerke aus Abschnitt 4.2 können statt mit zwei Elemente mit dem Merge von 32 Elementen beginnen, wenn die Elemente in sortierten Sechzehnergruppen angeordnet sind. Die Sortierung von 16 Elementen kann durch die Sortiernetzwerke in Abb. 4.4 erfolgen.

[7] Baddar und Batcher (2011, S. 27 ff.) erkennen nach Umsortierung der Vergleichsmodule die Teile-und-herrsche-Strategie für das Sortiernetz aus Abb. 4.4b.

4.4 Vektorisierungsstrategien für Sortiernetzwerke

4.4.1 Schrittbasierte Strategie

Wahl des passenden Sortiernetzwerks

Die schrittbasierte Vektorisierungsstrategie führt verschiedene Vergleichsmodule eines parallelen Schritts gleichzeitig aus. Für die Performanz der Strategie sind die Anzahl der parallelen Schritte und die Anordnung der Vergleichsmodule im Netzwerk entscheidend. Die Vergleichsmodule sind günstig angeordnet, wenn (1) das Vektorregister entsprechend seiner Kapazität die maximale Anzahl an Elementen sortiert, (2) überwiegend Shuffle- statt Permutations-Befehle[8] benutzt werden, und (3) die Vergleichsoperationen bei einem parallelen Schritt nicht gleichzeitig innerhalb und außerhalb des Vektorregisters stattfinden.

Insbesondere Punkt 3 schränkt die Auswahl passender Netzwerke zur Vektorisierung mittels der schrittbasierten Strategie ein. Die besten Sortiernetzwerke aus Abb. 4.4 auf Seite 27 sind mit Vektorregistern, die 8 Elemente aufnehmen, mit dieser Strategie nicht effizient realisierbar. Ab dem fünften parallelen Schritt gibt es Module, die vollständig im Vektorregister liegen, während andere beide Register überspannen. Die Implementierung von Odd-Even Mergesort aus Abb. 4.3b auf Seite 26 würde ab dem neunten parallelen Schritt ebenfalls Punkt 3 verletzen. Ein Vektorregister, das 16 Elemente aufnimmt, könnte dagegen diese mit 9 parallelen Schritten mit dem Netzwerk aus Abb. 4.4b auf Seite 27 sortieren, wobei dann evtl. vermehrt zwischen den 128-Bit-*Lanes* eines Registers getauscht wird, was Punkt 2 verletzt.[9]

Die Anzahl an Vergleichsmodulen ist bei der schrittbasierten Strategie erst dann relevant, wenn durch weniger Module innerhalb eines parallelen Schritts Freiräume entstehen, die keiner Bearbeitung durch ein Vektorregister bedürfen. Dies ist beim Odd-Even Mergesort ab der Sortierung von 32 Elementen der Fall (siehe zwölften parallelen Schritt in Abb. A.3 auf Seite 74). Da Odd-Even Mergesort gegen Punkt 3 verstößt, wird zur schrittbasierten Vektorisierung Bitonic Mergesort verwendet.

Vektorisierung eines Sortiernetzwerks für acht Elemente

In Abb. 4.5 auf der nächsten Seite ist ein Hybridnetzwerk dargestellt. Die beiden Vierergruppen werden mittels Odd-Even Mergesort sortiert (Schritte 1 bis 3). Ab dem

[8] Mit Permutation können Elemente innerhalb eines Vektors beliebig vertauscht werden, beim Shuffle lediglich innerhalb der 128-Bit-*Lanes* eines Vektorregisters (vgl. Abschnitt 2.3 auf Seite 17).

[9] Dies ist abhängig von Datentyp und Registerbreite. Es können beispielsweise 16 8-Bit-Integer innerhalb einer 128-Bit-*Lane* sortiert werden, wodurch der Tausch zwischen den 128-Bit-*Lanes* entfällt.

vierten parallelen Schritt wird Bitonic Mergesort verwendet. Lediglich Schritt vier erfordert eine Permutation. Die übrigen Schritte benutzen Shuffle-Instruktionen. Mit Odd-Even Mergesort wären Permutationen für die beiden letzten Schritte notwendig.

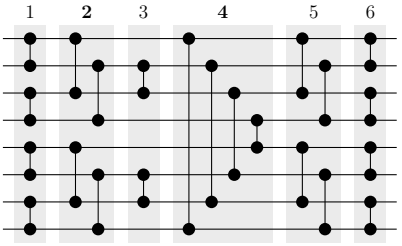


Abb. 4.5: Hybridnetzwerk für 8 Elemente (22 Module, 6 Schritte)

Eine hinsichtlich der Latenz nicht optimierte vektorisierte Implementierung des Hybridnetzwerks ist in Quelltext 4.2 enthalten. Die Makro-Funktion `SORT_8` sortiert einen `__m256i` Vektor mit acht Integern. `COEX_PERMUTE` simuliert einen parallelen Schritt auf dem Vektor. Hierfür werden die Indices zur Permutation (`a` bis `h`) und eine 8-Bit-Maske ergänzend zum Vektor beim Aufruf von `COEX_PERMUTE` angegeben. Entsprechend der Indices wird in Zeile 4 eine permutierte Kopie des Vektors, in der die Werte innerhalb der Module vertauscht sind, erstellt. Zwischen den beiden Vektoren wird das paarweise Minimum und Maximum der Elemente berechnet. In Zeile 7 werden schließlich die Vektoren mit dem paarweisen Minimum bzw. Maximum entsprechend der 8-Bit-Maske vermischt. Die Maske gibt, an welche Elemente aus dem Minimum-Vektor und welche aus dem Maximum-Vektor extrahiert werden.

Quelltext 4.2: Sortiernetzwerk für 8 Elemente ohne Optimierung

```
1  /* vektorisiertes compare-exchange mit Permutation */
2  #define COEX_PERMUTE(vec, a, b, c, d, e, f, g, h, MASK){                  \
3    __m256i permute_mask = _mm256_setr_epi32(a, b, c, d, e, f, g, h);       \
4    __m256i permuted = _mm256_permutevar8x32_epi32(vec, permute_mask);      \
5    __m256i min = _mm256_min_epi32(permuted, vec);                          \
6    __m256i max = _mm256_max_epi32(permuted, vec);                          \
7    vec = _mm256_blend_epi32(min, max, MASK);}
8
9  #define SORT_8(vec){                        /* sortiere aufsteigend 8 int */ \
10   COEX_PERMUTE(vec, 1, 0, 3, 2, 5, 4, 7, 6, 0b10101010);  /* Schritt 1 */ \
11   COEX_PERMUTE(vec, 2, 3, 0, 1, 6, 7, 4, 5, 0b11001100);  /* Schritt 2 */ \
12   COEX_PERMUTE(vec, 0, 2, 1, 3, 4, 6, 5, 7, 0b01000100);  /* Schritt 3 */ \
13   COEX_PERMUTE(vec, 7, 6, 5, 4, 3, 2, 1, 0, 0b11110000);  /* Schritt 4 */ \
14   COEX_PERMUTE(vec, 2, 3, 0, 1, 6, 7, 4, 5, 0b11001100);  /* Schritt 5 */ \
15   COEX_PERMUTE(vec, 1, 0, 3, 2, 5, 4, 7, 6, 0b10101010);} /* Schritt 6 */
```

Im Quelltext A.1 auf Seite 75 ist ein vollständiges Programmbeispiel zur vektorisierten Sortierung von acht Elementen mit dem Hybridnetzwerk dargestellt. Im Unterschied zu Quelltext 4.2 verwendet die Variante im Anhang überwiegend Shuffle-Instruktionen und errechnet die 8-Bit-Masken zur Kompilierzeit aus den Indices

der Permutationen.[10] Ebenfalls wird gezeigt, wie mit der gleichen Makro-Funktion `SORT_8` sowohl aufsteigend als auch absteigend sortiert werden kann.

Vektorisierung von Bitonic Mergesort für 16 Elemente

Quelltext 4.3 demonstriert, wie die vektorisierte Sortierung von 16 Integern mit einem Sortiernetzwerk erfolgen könnte. Als Input bekommt `SORT_16` in Zeile 18 zwei Vektoren. Die beiden Vektoren werden aufsteigend sortiert und miteinander gemergt. Für den Merge werden die Schritte 7 bis 10 des bitonischen Sortiernetzwerks aus Abb. 4.3a auf Seite 26 benutzt. Zur Realisierung des siebten parallelen Schrittes wird der zweite Vektor in Zeile 14 umgekehrt. Die Makro-Funktion `COEX_VERTICAL` führt anschließend eine paarweise Vertauschung der Elemente zwischen den beiden Vektoren durch, falls diese im ersten Vektor größer als im zweiten sind. Zur Ausführung der letzten drei Schritte des Netzwerks wird auf den beiden Vektoren die Makro-Funktion `LAST_3_STEPS` verwendet.

Quelltext 4.3: Sortiernetzwerk für 16 Elemente ohne Optimierung

```
1  #define REVERSE_VEC(v){ /* kehre Reihenfolge der Zahlen im Vektor um */   \
2    v = _mm256_permutevar8x32_epi32(v,                                      \
3            _mm256_setr_epi32(7, 6, 5, 4, 3, 2, 1, 0));}
4
5  #define COEX_VERTICAL(a, b){ /* berechne acht Vergleichsmodule */         \
6    __m256i c = a; a = _mm256_min_epi32(a, b); b = _mm256_max_epi32(c, b);}
7
8  #define LAST_3_STEPS(v){ /* letzte drei Schritte des Netzwerks */         \
9    COEX_PERMUTE(v, 4, 5, 6, 7, 0, 1, 2, 3, 0b11110000);                    \
10   COEX_PERMUTE(v, 2, 3, 0, 1, 6, 7, 4, 5, 0b11001100);                    \
11   COEX_PERMUTE(v, 1, 0, 3, 2, 5, 4, 7, 6, 0b10101010);}
12
13 #define MERGE_16(v1, v2){ /* merge zwei sortierte Vektoren */             \
14   REVERSE_VEC(v2);                                                        \
15   COEX_VERTICAL(v1, v2); /* Schritt 7 */                                  \
16   LAST_3_STEPS(v1); LAST_3_STEPS(v2);} /* Schritte 8, 9 und 10 */
17
18 #define SORT_16(v1, v2){ /* sortiere 16 int */                           \
19   SORT_8(v1); SORT_8(v2); /* sortiere Vektoren v1 und v2 */               \
20   MERGE_16(v1, v2);} /* merge v1 und v2 */
```

Die beiden Makro-Funktionen `COEX_PERMUTE` und `COEX_VERTICAL` simulieren vektorisiert Vergleichsmodule. `COEX_VERTICAL` benötigt die Instruktionen paarweises Minimum und paarweises Maximum, um acht Vergleichsmodule zu berechnen, während `COEX_PERMUTE` ergänzend zu diesen einer Permutation und eines *Blend*-Befehls bedarf, und lediglich vier Vergleichsmodule simuliert. `COEX_VERTICAL` ist somit

[10] Bramas (2017) sortiert ähnlich wie in Quelltext 4.2 auf Seite 29 lediglich mit Permutations-Instruktionen, wobei AVX-512 ebenfalls Shuffle-Instruktionen wie `_mm512_shuffle_epi32` bereitstellt (siehe Intrinsics Guide 2018).

effizienter bezüglich der benötigten Rechenzeit pro simulierten Modul. Mit jeder Verdoppelung der Anzahl zu mergender Vektoren kommt ein neuer paralleler Schritt im bitonischen Netzwerk hinzu. Dieser Schritt kann vollständig mit `COEX_VERTICAL` Funktionen implementiert werden.

Die Permutation in Zeile 14 ist unnötig, falls der zweite Vektor absteigend sortiert wird. Ferner können die beiden letzten Permutationen in `LAST_3_STEPS` durch Shuffle-Instruktionen ersetzt werden. Der Quelltext A.2 auf Seite 76 im Anhang stellt eine Variante des Sortiernetzwerks mit diesen beiden Optimierungen dar.

Algorithmisierung von Bitonic Mergesort

Durch die oben angeführten drei Makro-Funktionen `COEX_PERMUTE`, `REVERSE_VEC` und `COEX_VERTICAL` ist es möglich, bitonische Sortiernetzwerke für beliebige N Vektoren zu erstellen. Zum Unterscheiden wird N für die Anzahl der Vektoren und n für die Menge der zu sortierten Elemente verwendet. Soweit nicht anders vermerkt gilt $n = 8 \cdot N$. Statt die Netzwerke für alle benötigten Längen einzeln zu erstellen, kann die Berechnung dieser algorithmisiert werden. Ein vektorisierter Algorithmus für Bitonic Mergesort ist in Quelltext 4.4 dargelegt.

Quelltext 4.4: Bitonic Mergesort ohne Optimierung

```
1  /* N Vektoren sortieren, d. h. insgesamt  N * 8 int */
2  inline void sort_bitonic(__m256i *vecs, const int N){
3    for (int i = 0; i < N; ++i) SORT_8(vecs[i]); /* sortiere Achtergruppen */
4    for (int t = 2; t < N * 2; t *= 2){ /* merge 2, dann 4, 8 ... Vektoren */
5      for (int l = 0; l < N; l += t){
6        for (int j = std::max(l + t - N, 0); j < t/2 ; j += 1){
7          REVERSE_VEC(vecs[l + t - 1 - j]);
8          COEX_VERTICAL(vecs[l + j], vecs[l + t - 1 - j]);}}
9      for (int m = t / 2; m > 1; m /= 2){
10       for (int k = 0; k < N - m / 2; k += m){
11         const int bound = std::min((k + m / 2), N - (m / 2));
12         for (int j = k; j < bound; j += 1){
13           COEX_VERTICAL(vecs[j], vecs[m / 2 + j]);}}}
14     for (int i = 0; i < N; i += 1){
15       LAST_3_STEPS(vecs[i]);}}}
```

Die Funktion `sort_bitonic` erhält als Eingabe einen Pointer `vecs` auf das zu sortierende Array aus Vektoren und die Anzahl der Vektoren `N` darin. In Zeile 3 werden die acht Integer jedes Vektors aufsteigend sortiert. In der nächsten Zeile beginnt die Merge-Phase. Die Variable `t` gibt an, wie viele Vektoren bei der jeweiligen Iteration zu mergen sind. In der ersten Iteration werden jeweils zwei Vektoren gemergt, in der nächsten vier usw. Die For-Schleife in Zeile 5 iteriert mit Inkrement von `t`, d. h. der Anzahl zu mergender Vektoren über alle Vektoren des Vektor-Arrays und führt den ersten parallelen Schritt des Merges aus. Für `t = 2` ist es Schritt 7 aus

Abb. 4.3a auf Seite 26, für `t = 4` ist es Schritt 11 aus Abb. A.2 auf Seite 73. Die dreifach geschachtelte For-Schleife, die in Zeile 5 beginnt, berechnet die übrigen parallelen Schritte außer den letzten drei. Die letzte For-Schleife von `sort_bitonic` iteriert nochmals über alle Vektoren und führt die drei letzten parallelen Schritte aus. Die Berechnungen mit `std::max` und `std::min` innerhalb von `sort_bitonic` sorgen dafür, dass die Anzahl an Vektoren `N` keine Zweierpotenz sein muss.

Die Funktion `sort_bitonic` kann für größere bzw. kleinere Vektorregister und andere Datentypen angepasst werden. Beispielsweise enthielte ein Vektorregister 32 Elemente, dann müssten in Zeile 3 in jeden Vektor 32 Elemente sortiert werden und die letzte For-Schleife der Funktion würde die letzten fünf parallelen Schritte berechnen. Die Funktion, wie sie hier dargestellt ist, erfordert ausgerichteten Speicher und dass die Anzahl zu sortierter Elemente n ohne Rest durch acht teilbar ist. Falls diese Annahmen nicht zutreffen, dann könnten die Elemente in ein ausgerichtetes Puffer-Array kopiert und sortiert werden. Das Puffer-Array könnte zur nächsten Sortierung wiederverwendet werden.

Optimierung der Modul-Auslastung

Ein Vergleichsmodul besteht schematisch aus zwei Modulknoten und einer Verbindungslinie zwischen den Knoten (siehe Abb. 4.1b auf Seite 23). Die Elemente der beiden Knoten werden innerhalb des Moduls sortiert. Die Makro-Funktionen `COEX_PERMUTE` bzw. `COEX_SHUFFLE` (im Anhang) berechnen gleichzeitig vier Vergleichsmodule, da die Modulknoten eines Moduls sich im gleichen Vektorregister befinden. Optimal für AVX2 ist die gleichzeitige Berechnung von acht Vergleichsmodulen. Dies ist möglich, wenn die Modulknoten der einzelnen Module in zwei unterschiedlichen Registern die gleichen Positionen besetzen. Mit steigender Anzahl an Elementen gestattet die Struktur von Bitonic Mergesort, vermehrt acht Vergleichsmodule gleichzeitig auszuführen. Deshalb dominiert im vektorisierten Bitonic Mergesort Algorithmus in Quelltext 4.4 auf Seite 31 mit steigender Anzahl an Vektoren N die Makro-Funktion `COEX_VERTICAL`, die gleichzeitig acht Vergleichsmodule durch paarweise Vertauschung der Elemente zwischen den beiden Vektoren berechnet. Im Folgenden wird eine Technik vorgestellt, die durch Umsortierung der Modulknoten die Verwendung von `COEX_VERTICAL` bei jedem parallelen Schritt ermöglicht und damit die Makro-Funktionen `COEX_PERMUTE` bzw. `COEX_SHUFFLE` ersetzt.

Zur Vereinfachung sei die Kapazität von Vektoren auf vier Elemente beschränkt. Die Eingabe besteht aus zwei vollbesetzten Vektoren, d. h. acht Elementen. Als Sortiernetzwerk wird Bitonic Mergesort verwendet. In Abb. 4.6 auf der nächsten Seite ist die Vorgehensweise zur Sortierung der acht Elemente dargelegt. Im ersten Schritt hat

Bitonic Mergesort für acht Elemente die Vergleichsmodule $\{(1,2),(3,4),(5,6),(7,8)\}$, will heißen, dass das erste und zweite Element miteinander verglichen werden, das dritte Element mit dem vierten usw. (vgl. Abb. A.1 auf Seite 73). Welche Position im Vektor mit welchen Modulknoten zu Beginn der Sortierung in Verbindung gebracht wird, ist frei wählbar.

Die Sortierung startet unter der Annahme, dass der erste Vektor die Knoten $\{1,3,5,7\}$ repräsentiert und der zweite $\{2,4,6,8\}$. Die Modulknoten befinden sich in Abb. 4.6 oberhalb bzw. unterhalb der Vektoren. Im Schritt 1 kann ohne Anpassung der Vektoren direkt die Operation `COEX_VERTICAL` zwischen den beiden Vektoren ausgeführt werden. Die Positionen der Module sind nach dem ersten Schritt unverändert.

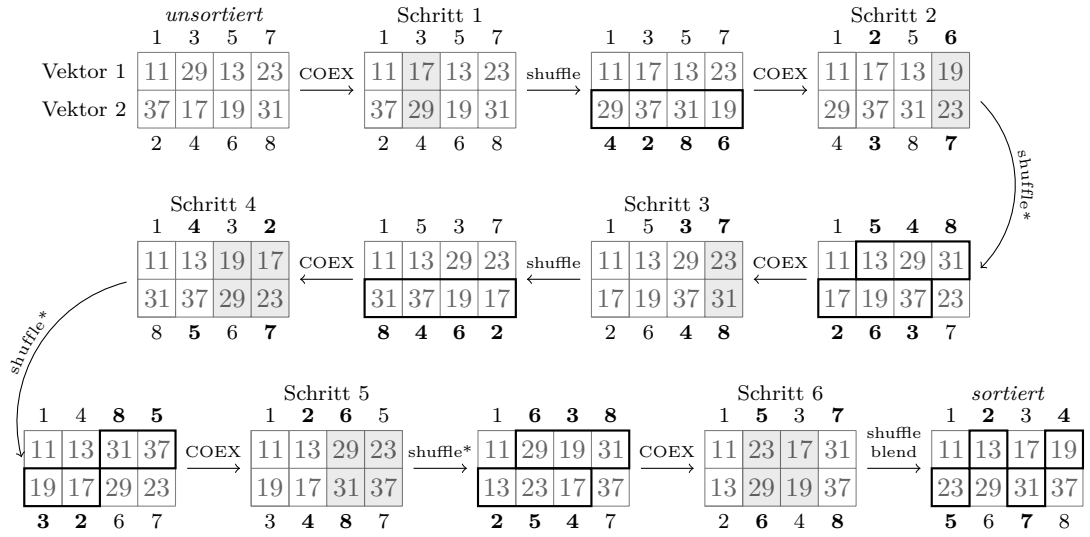


Abb. 4.6: Optimierung der Modul-Auslastung in Vektorregistern

Der zweite parallele Schritt benötigt die Module $\{(1,4),(2,3),(5,8),(6,7)\}$. Vor Ausführung von Schritt 2 wird der zweite Vektor geshuffelt, sodass die erforderlichen Modulknoten einander gegenüberstehen. Die Operation `COEX_VERTICAL` in Schritt 2 vertauscht die fettgedruckten Module (3 mit 2 und 7 mit 6). Diese Vertauschung ist notwendig, da das Minimum zwischen zwei Modulknoten nach Ausführung von `COEX_VERTICAL` immer im ersten Vektor ist und das Minimum erhält der Modulknoten mit der niedrigeren Nummer.

Im dritten Schritt werden die Module $\{(1,2),(3,4),(5,6),(7,8)\}$ verwendet. Mit zwei Shuffle Operationen, die jeweils zwei Vektoren miteinander mischen, werden die Modulknoten für den dritten Schritt vorbereitet.[11] Die passenden Modulknoten

[11] Die Shuffle Operationen, die zwei Vektoren als Input erhalten und diese entsprechend einer Maske zu einem Vektor vermischen, sind in Abb. 4.6 als shuffle* gekennzeichnet.

stehen einander gegenüber und es wird der dritte Schritt mittels `COEX_VERTICAL` vollzogen. Beim dritten Schritt erfolgt der Tausch der Modulknoten 4 mit 3 und 8 mit 7, da nach einer `COEX_VERTICAL` Operation der erste Vektor immer den kleineren der beiden Modulknoten innerhalb eines Moduls erhält.

Für den vierten Schritt ist ein Shuffle des zweiten Vektors notwendig, damit die Module $\{(1,8),(2,7),(3,6),(4,5)\}$ mit `COEX_VERTICAL` realisiert werden können usw. Nach Ausführung von Schritt 6 sind in Vektor 1 immer die Elemente der Modulknoten $\{1,5,3,7\}$ und in Vektor zwei $\{2,6,4,8\}$ enthalten. Nach der Neuanordnung der Elemente mit zwei Shuffle und zwei Blend Instruktionen entsprechend den Modulknoten sind die acht Elemente sortiert.

Die Implementierung des eben beschriebenen Beispiels ist im Quelltext A.3 auf Seite 76 dargelegt. Ergänzend veranschaulicht Quelltext A.4 auf Seite 77 mit der Technik der optimalen Modul-Auslastung die Sortierung von 16 Integern mit zwei 256-Bit-Vektoren. Für die Implementierung der Netzwerke mit diesem Verfahren wird zusätzlich zu den in Abschnitt 2.3 auf Seite 17 vorgestellten intrinsischen Funktionen eine Schuffle-Instruktion zum Vermischen von zwei Vektoren verwendet. Die Shuffle-Instruktion für zwei Vektoren ist mit einer Latenz und Durchsatz von einem Taktzyklus genauso effizient wie die Shuffle-Instruktion für einen Vektor. Die Funktionsweise dieser Instruktion ist im Quelltext 4.5 demonstriert.

Quelltext 4.5: Shuffle von zwei Vektoren

```
1 __m256i v1 = _mm256_setr_epi32(1, 2, 3, 4, 5, 6, 7, 8);
2 __m256i v2 = _mm256_setr_epi32(9, 10, 11, 12, 13, 14, 15, 16);
3 __m256i tmp = v1; /* ersten Vektor zwischenspeichern */
4 v1 = SHUFFLE_2_VECS(v1, v2, 0b10001000)  /* v1 ist {1 3  9 11 5 7 13 15} */
5 v2 = SHUFFLE_2_VECS(tmp, v2, 0b11011101) /* v2 ist {2 4 10 12 6 8 14 16} */
```

Die Makro-Funktion `SHUFFLE_2_VECS` reinterpretiert zur Kompilierzeit die Integer-Vektoren in Fließkommazahlen-Vektoren um, damit die intrinsische Funktion `_mm256_shuffle_ps` verwendet werden kann (vgl. Quelltext A.4 auf Seite 77). Die 8-Bit-Kompilierzeitkonstante beschreibt von rechts nach links zwei-bitweise lesend, wie die Vektoren innerhalb der 128-Bit-*Lanes* zu vermischen sind. Die ersten 4 Bit von rechts repräsentieren die zwei Indices der Werte, die aus `v1` verwendet werden, die letzten 4 Bit sind die Indices der Werte, die aus `v2` entnommen werden. Die Maske `10 00 10 00` bedeutet, dass aus `v1` der erste und dritte und aus `v2` ebenfalls der erste und dritte Wert innerhalb der 128-Bit-*Lanes* in den Ergebnisvektor zu shuffeln sind.

4.4.2 Modulbasierte Strategie

Wahl des passenden Sortiernetzwerks

Die modulbasierte Vektorisierungsstrategie von Sortiernetzwerken ist in zwei Phasen unterteilt. Das zu sortierende Array wird als Matrix in zeilendominierter Reihenfolge (*row-major order*) aufgefasst. In AVX2 für den Datentyp `int` und N Eingabevektoren ergibt sich eine Matrix mit N Zeilen und acht Spalten. In der ersten Phase werden die Elemente Spaltenweise sortiert. In der zweiten Phase wird die Matrix transponiert und die acht sortierten Blöcke gemergt (vgl. Chhugani et al. 2008, S. 1317). In Abb. 4.7 ist die Vorgehensweise anhand von vier Vektoren mit jeweils vier Elementen dargestellt.

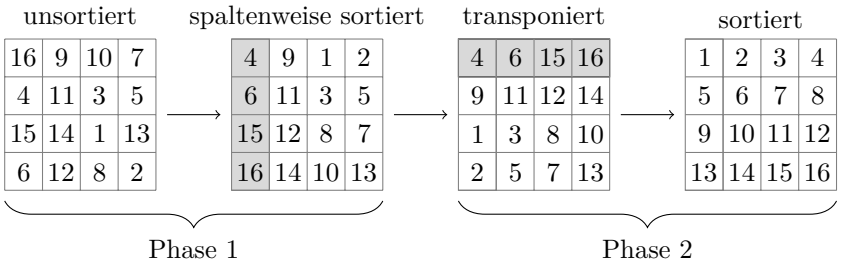


Abb. 4.7: Modulbasierte Vektorisierungsstrategie mit Transposition

Die Matrix muss nicht quadratisch sein. Die vektorisierte Operation für die Implementierung der ersten Phase ist `COEX_VERTICAL`, welche in Abschnitt 4.4.1 auf Seite 30 vorgestellt wurde. `COEX_VERTICAL` führt acht Vergleichsmodule gleichzeitig aus und verwendet die Instruktionen paarweises Minimum und Maximum. Die Bezeichnung „modulbasiert“ für die Vektorisierungsstrategie resultiert daher, dass in der ersten Phase dasselbe Vergleichsmodul auf allen Elementen des SIMD-Vektors ausgeführt wird. Die Anzahl an Modulen im Netzwerk bestimmt die Anzahl benötigter Instruktionen. Für die erste Phase sind insbesondere Netzwerke mit minimaler Anzahl an Vergleichsmodulen geeignet. Zur Sortierung von 16 Vektoren kann das Netzwerk mit 60 Modulen aus Abb. 4.4a auf Seite 27 verwendet werden. Zum Mergen von mehr als 16 Vektoren in der ersten Phase ist Odd-Even Merge geeignet.

Sind die Elemente spaltenweise sortiert, beginnt die zweite Phase. Eine Möglichkeit die zweite Phase zu realisieren ist wie in Abb. 4.7 gezeigt mit Transposition der Matrix und anschließenden Merge-Operationen. Zum Mergen kann Bitonic Merge ähnlich wie bei der schrittbasierten Vektorisierungsstrategie aus den vorherigen Abschnitt verwendet werden. Eine Implementierung ohne Transposition mit direktem Merge der Spalten ist ebenfalls möglich und wird nach der Behandlung der Variante mit Transposition beschrieben.

Modulbasiert mit Transposition

Die spaltenweise Sortierung der ersten Phase von N Vektoren ist algorithmisch identisch mit der skalaren Sortierung von n Elementen. Die skalare Funktion in Quelltext 4.1 auf Seite 24 sortiert vier Integer, während die vektorisierte in Quelltext 4.6 in jeder der acht Spalten vier Integer sortiert.

Quelltext 4.6: Sortierung von 8 Spalten mit jeweils 4 Elementen

```
1 #define COEX_VERTICAL(a, b){  /* berechne acht Vergleichsmodule */        \
2   __m256i c = a; a = _mm256_min_epi32(a, b); b = _mm256_max_epi32(c, b);}
3 /* Netzwerk um in 8 Spalten 4 Integer zu sortieren */
4 inline void sort_4_in_8_columns(__m256i *vecs){
5   COEX_VERTICAL(vecs[0], vecs[1]); COEX_VERTICAL(vecs[2], vecs[3]); /*S1*/
6   COEX_VERTICAL(vecs[0], vecs[2]); COEX_VERTICAL(vecs[1], vecs[3]); /*S2*/
7   COEX_VERTICAL(vecs[1], vecs[2]);}                                 /*S3*/
```

Inoue, Moriyama et al. benutzen für die erste als auch für die zweite Phase Combsort, eine verbesserte Variante von Bubblesort mit einer Durchschnittskomplexität von $\mathcal{O}(n \log n)$, im Worst-Case jedoch $\mathcal{O}(n^2)$. Als weiterer datenabhängiger Sortieralgorithmus eignet sich Shellsort für die erste Phase. In Quelltext A.5 auf Seite 79 ist eine Variante von Shellsort mit der von Ciura (2001) ermittelten Sequenz zur Minimierung der Vergleichsoperationen dargestellt. Die auf den Datentyp `__m256i` angepasste Funktion sortiert spaltenweise eine beliebige Anzahl von N Vektoren.

Statt eines datenabhängigen Sortierverfahrens verwenden Chhugani et al. (2008) Odd-Even Mergesort für die erste Phase und nach der Transponierung Bitonic Merge in der zweiten Phase. Eine zusätzliche Reduktion der Modulanzahl kann durch das in Abb. 4.4a auf Seite 27 gezeigte Netzwerk mit 60 Modulen für 16 Elemente erreicht werden. Die Implementierung des 60-Modul-Netzwerks und die darauffolgende Transposition sind im Quelltext A.6 auf Seite 79 enthalten. Eine vollständige Transposition der 128 Elemente[12] ist nicht notwendig. Es ist ausreichend zweimal 64 Elemente zu transponieren und acht Vektoren zu vertauschen, damit sich acht Zeilen mit jeweils 16 sortierten Elementen ergeben.[13] Weshalb eine blockweise Transposition der nicht quadratischen Matrix mit anschließender Vertauschung der Vektoren ausreichend ist, zeigt Abb. 4.8 auf der nächsten Seite. Die acht Vektoren mit jeweils vier Elementen liegen spaltenweise sortiert vor. Nach der Transposition der oberen vier und der unteren vier Vektoren werden zwei Vertauschungen ausgeführt. Nach den Vertauschungen entstehen vier Zeilen mit jeweils 8 sortierten Elementen.

[12] Die 128 Elemente ergeben sich aus 8 Elementen pro Vektor und insgesamt 16 Vektoren die spaltenweise sortiert werden.

[13] Die Instruktionsreihenfolge für die Transponierung der 64 Integer in Quelltext A.6 auf Seite 79 stammt von Intel Corporation (2016a, S. 11-29 f.).

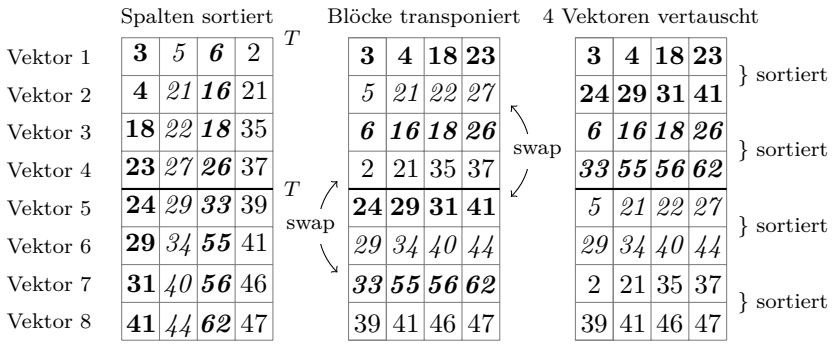


Abb. 4.8: Blockweise Transposition und Vertauschung der Vektoren

Diese Vertauschungsstrategie ist lediglich für kleine N effizient, da beispielsweise für 32 spaltenweise sortierte Vektoren, nach acht einzelnen Transpositionen bereits 24 Vertauschungsoperationen notwendig wären, um vier Zeilen mit jeweils 32 sortierten Elementen herzustellen.

Modulbasiert ohne Transposition

Die zweite Phase mergt die Spalten zu einer in zeilendominierter Reihenfolge sortierten Matrix. Einerseits kann der Merge nach der Transposition ähnlich wie bei der schrittbasierten Strategie erfolgen, andererseits ist es möglich, ohne Transposition die Spalten mit Bitonic Merge zu mergen. Zu Beginn der zweiten Phase sind die Spalten sortiert. In Abb. 4.9 auf der nächsten Seite sind innerhalb der Kreise die zu den Werten gehörenden Modulknoten visualisiert.[14] Die tatsächlichen Zahlen, die sortiert werden, sind nicht dargestellt. Durch Shuffle-Instruktionen werden die Modulknoten für Vergleichs-und-Austausch-Operationen (COEX) günstig angeordnet. Da im Beispiel bereits die Vierergruppen in den Spalten sortiert sind, beginnt die zweite Phase mit Schritt 4 des bitonischen Sortiernetzwerks für 16 Elemente aus Abb. 4.3a auf Seite 26.

Die Vergleichsmodule für den vierten Schritt sind $\{(1,8),(2,7),(3,6),(4,5),(9,16),(10,15),(11,14),(12,13)\}$. Nach den Shuffle der Vektoren 3 und 4 sind die Modulknoten für die COEX-Operationen im Schritt 4 vorbereitet. Nach Ausführung der zwei Vergleichs-und-Austausch-Operationen werden die Vektoren 2 und 4 geshuffelt, damit im fünften Schritt die Module $\{(1,3),(2,4),(5,7),(6,8),(9,11),(10,12),(13,15),(14,16)\}$ realisiert werden. Im sechsten Schritt erfolgt ein paarweiser Vergleich der Nachbarwerte. Zur Vorbereitung von Schritt 7

[14] Zur Vereinfachung ist das Beispiel in Abb. 4.9 auf der nächsten Seite für Vektoren mit vier Elementen und einer quadratischen Matrix dargestellt.

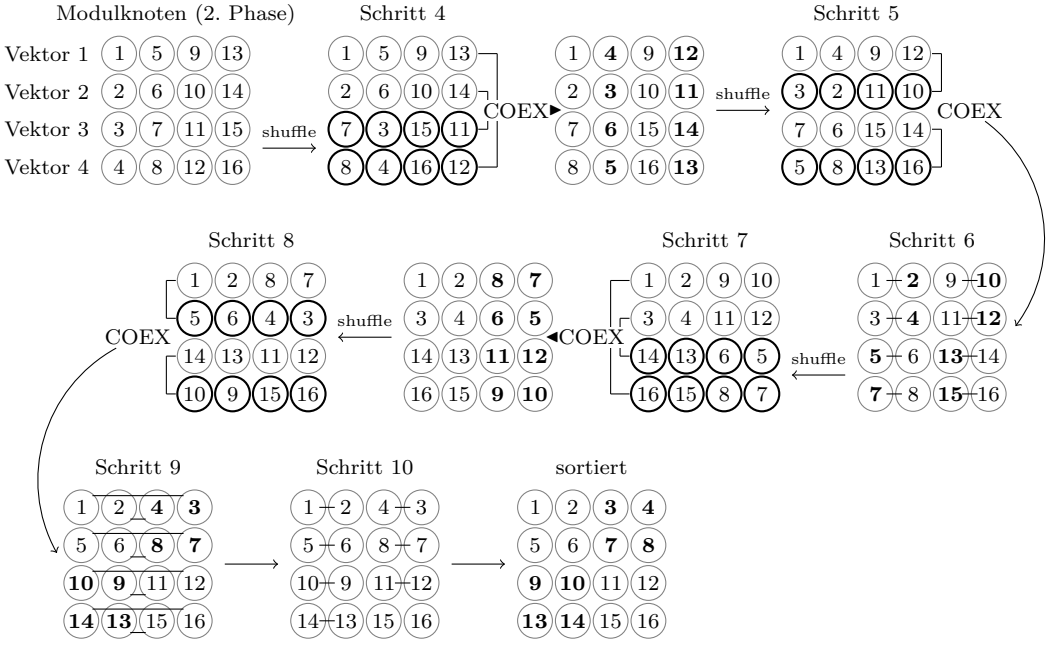


Abb. 4.9: Bitonic Merge für sortierte Spalten

werden die Vektoren 3 und 4 mit den Shuffle umgekehrt usw. Nach einem paarweisen Vergleich der Nachbarwerte im Schritt 10 sind die 16 Elemente sortiert.

Diese Strategie zum Mergen der sortierten Spalten kann auf Vektoren mit acht Elementen und nicht quadratische Matrizen ausgeweitet werden. Im Quelltext A.7 auf Seite 81 werden zuerst die acht Spalten mit den 60-Modul-Netzwerk in Sechzehnergruppen sortiert und anschließend mit Odd-Even Merge gemergt.[15] Für den Merge der sortierten Spalten wird wie in Abb. 4.9 dargelegt Bitonic Merge ohne Transposition verwendet.

4.4.3 Performanz der Vektorisierungsstrategien

Gegenstand dieses Unterabschnitts ist die Ermittlung der Performanz von drei Vektorisierungsstrategien von Sortiernetzwerken im Vergleich zu `std::sort`. Die einzelnen Strategien sind wie folgt implementiert:

schrittbasiert naiv: Bitonic Mergesort ohne Optimierungen aus Quelltext 4.4 auf Seite 31

schrittbasiert optimiert: zeilenweise Sechzehnergruppen mit Quelltext A.4 auf Seite 77 sortieren und mit optimiertem Bitonic Merge aus Quelltext A.8 auf Seite 83 mergen (jede Iteration der For-Schleife berechnet 16 statt 8 Module)

[15] Mit dem vektorisierten Odd-Even Mergesort aus Quelltext A.7 auf Seite 81 können ebenfalls die kompletten Spalten, ohne vorher das 60-Modul-Netzwerk zu benutzen, sortiert werden.

schritt- und modulbasiert optimiert: (1) spaltenweise Sechzehnergruppen mit 60-Modul-Netzwerk aus Quelltext A.6 auf Seite 79 sortieren und diese ohne Transposition (wie in Abb. 4.9 auf Seite 38) zu zeilenweise sortierten 128 Elementen (entspricht 16 Vektoren) mergen[16], (2) restlichen N mod 16 Vektoren wie bei *schrittbasiert optimiert* sortieren, (3) optimierten Bitonic Merge Algorithmus[17] für sortierte Gruppen von 16 Vektoren verwenden

In Abb. 4.10 sind die unterschiedlichen Speedups der drei Strategien ggü. `std::sort` in Abhängigkeit von der Anzahl zu sortierender Integer n abgetragen. Die Arrays für die Sortierung sind mit gleichverteilten Zufallszahlen befüllt und an keiner bestimmten Speichergrenze ausgerichtet, sodass die Kopierzeiten in und aus den ausgerichteten Buffer mit in die Ermittlung des Speedups einfließen. Beginnend bei $n = 16$ wird die Anzahl der Werte im Array so lange mit 640 inkrementiert, bis $n = 10\,896$ gilt. Insgesamt sind für jede Strategie 18 Werte abgebildet.

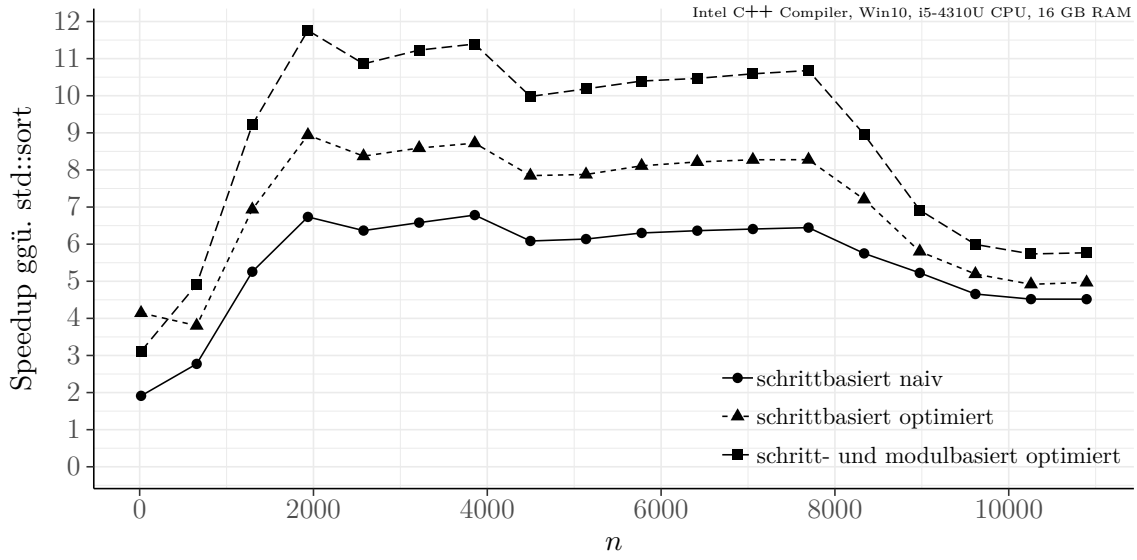


Abb. 4.10: Speedup unterschiedlicher Vektorisierungsstrategien ggü. std::sort

Bis etwa 2 000 Elemente wächst der Speedup. Mit steigender Anzahl der Werte im Array benötigt ein Vergleichsmodul im Durchschnitt weniger Instruktionen, da vermehrt allein `COEX_VERTICAL` und kein Shuffle bzw. Permutation zur Berechnung von Modulen eingesetzt wird. Ebenfalls werden längere For-Schleifen ausgeführt, was die Kosten der verschachtelten Schleifen im Bitonic Mergesort Algorithmus reduziert. Am performantesten sind die drei Implementierungen für den Bereich von

[16] Code für den Merge der Spalten ist durch Konsolenausgabe der Instruktionen von Phase 2 der Funktionen `no_transpose_sort` aus Quelltext A.7 auf Seite 81 automatisch generiert.

[17] Optimierung durch teilweises Entrollen der For-Schleife in Zeile 11 des Bitonic Merge Algorithmus von Quelltext A.8 auf Seite 83.

2 000 bis 4 000 Elemente. Die hybride Strategie *schritt- und modulbasiert optimiert* erreicht in diesem Bereich eine Beschleunigung um den Faktor 11 bis 11,7. Ab 8 000 Elemente fällt der Speedup und ist bei $n = 100\,000\,000$ in etwa 1 (nicht dargestellt in Abb. 4.10).[18]Der sinkende Speedup ist bedingt durch die höhere Zeitkomplexität der Sortiernetzwerke im Vergleich zu `std::sort` (siehe Abschnitt 4.2 auf Seite 25). Die Sortiernetzwerke allein reichen nicht aus, um große Arrays mit einem gleichbleibend hohen Speedup zu sortieren, jedoch können diese in andere vektorisierte Algorithmen integriert werden und damit die Gesamtperformanz eines hybriden Sortieralgorithmus steigern.

[18]Die Verringerung des Beschleunigungsfaktors mit steigender Anzahl der Elemente im Array kann verlangsamt werden, wenn ein Block von beispielsweise 8192 Elementen ganz sortiert wird, bevor die Sortierung mit dem nächsten Block beginnt. Bei der Sortierung in Blöcken wird vermehrt auf den Cache zugegriffen. Ebenfalls verlangsamt die rekursive Berechnung der Sortiernetzwerke (nicht behandelt in der Arbeit) durch bessere Wiederverwendung des Caches für größere n den Abfall des Speedups, für kleinere n ist sie jedoch langsamer als die in der Arbeit vorgestellte iterative Berechnung.

5 Mergesort

5.1 Skalarer Mergesort

Beim Mergesort wird eine Folge $f = a_1, a_2, \ldots$ in zwei möglichst gleich große Teilfolgen $f_1 = a_1, a_2, \ldots, a_{\lfloor n/2 \rfloor}$ und $f_2 = a_{\lfloor n/2 \rfloor + 1}, a_{\lfloor n/2 \rfloor + 2}, \ldots, a_n$ aufgeteilt. Die beiden Folgen werden durch rekursiven Aufruf von Mergesort erneut geteilt usw. Die Rekursion endet bei einelementigen oder leeren Teilfolgen, da diese sortiert sind. Anschließend beginnt der Merge der Teilfolgen. Mit einem Merge werden jeweils die beiden Teilfolgen des jeweiligen Mergesort-Aufrufs vereinigt. Der Algorithmus endet, sobald die beim ersten Aufruf aufgeteilten Teilfolgen mit den Merge zu einem geordneten Array zusammengefasst sind. Mergesort ist ein stabiles Sortierverfahren.[1] Mergesort benötigt zusätzlichen Speicher proportional zu n und hat eine Worst-Case Laufzeit von $\mathcal{O}(n \log n)$ (siehe Nebel 2012, S. 225).

In Quelltext 5.1 auf der nächsten Seite ist eine nicht optimierte Variante von Mergesort gezeigt.[2] Für die Mergeoperation wird in Zeile 5 eine For-Schleife verwendet. Der zusätzliche erforderliche Speicher der Größe n wird beim Aufruf von `merge_sort_scalar` in Zeile 11 zu Beginn der Sortierung als Zeiger im zweiten Argument der Funktion mitgegeben.

5.2 Vom skalaren zum vektorisierten Mergesort

5.2.1 Sortiernetzwerke im vektorisierten Mergesort

Der vektorisierte Mergesort ist ein Hybrid aus Sortiernetzwerken und den skalaren Mergesort. Er vereint die hohe Performanz der Sortiernetzwerke für kleinere n

[1] Ein Sortierverfahren ist stabil, wenn die Datensätze mit gleichem Schlüssel nach der Sortierung dieselbe relative Reihenfolge wie vor der Sortierung haben. Zum Beispiel zwei Datensätze A und B haben denselben Schlüssel. Im unsortierten Array liegt A vor B. Falls nach der Sortierung garantiert ist, dass A immer vor B liegt, dann ist das Sortierverfahren stabil. In C++ STL ist `std::stable_sort` stabil, während `std::sort` nicht stabil ist.

[2] Drei mögliche Optimierungen für Mergesort sind (vgl. Sedgewick 2014, S. 299 ff.): (1) Insertionsort für kleine Teilfolgen verwenden, (2) Merge überspringen, falls letzter Wert der ersten Teilfolge kleiner-gleich ersten Wert der zweiten Teilfolge ist, (3) Zeit für das Kopieren der Werte aus dem Hilfsarray durch ständigen Tausch der Rollen von Array und Hilfsarray während der Rekursion eliminieren.

Quelltext 5.1: Skalarer Mergesort

```
1  /* Arrays a und b mit Längen size_a und size_b nach Array c mergen */
2  inline void merge_scalar(int *a, int *b, int *c,
3                   const int size_a, const int size_b, const int size_c) {
4    int idx_a = 0, idx_b = 0;
5    for (int i = 0; i < size_c; ++i) {
6      if (idx_a == size_a) { c[i] = b[idx_b++]; }        /* a abgearbeitet */
7      else if (idx_b == size_b) { c[i] = a[idx_a++]; }   /* b abgearbeitet */
8      else { c[i] = (a[idx_a] < b[idx_b]) ? a[idx_a++] : b[idx_b++]; }}}
9
10 /* Array der Länge n sortieren, c ist zusätzlicher Speicher der Länge n */
11 inline void merge_sort_scalar(int *arr, int *c, int n) {
12   if(n > 1) {   /* Rekursionsende für einelementige oder leere Teilfolge */
13     const int size_a = n / 2;                        /* Länge Teilfolge a */
14     const int size_b = n - size_a;                   /* Länge Teilfolge b */
15     merge_sort_scalar(arr, c, size_a);   /* Teilfolgen a und b sortieren */
16     merge_sort_scalar(arr + size_a, c + size_a, size_b);
17     merge_scalar(arr, arr + size_a, c, size_a, size_b, n);     /* mergen */
18     memcpy(arr, c, sizeof(int) * n); }} /* Werte von c nach arr kopieren */
```

mit einer ebenfalls auf Sortiernetzen beruhenden Mergingstrategie für größere n.[3] Im Unterschied zu Sortiernetzen hat der Merge für größere n eine asymptotische Komplexität von $\mathcal{O}(n)$ statt $\mathcal{O}(n \log n)$. Insgesamt hat der vektorisierte Mergesort eine asymptotische Komplexität von $\mathcal{O}(n \log n)$ und benötigt wie der skalare Mergesort für eine effiziente Implementierung zusätzlichen Speicher der Größe n.

5.2.2 Vektorisierte Mergeoperation

Der vektorisierte Merge von zwei beliebig langen Arrays aus Vektoren benötigt ein Mergenetzwerk für mindestens zwei einzelne Vektoren. Unter der Annahme, dass ein Vektorregister 4 Elemente aufnimmt, ist zumindest ein Mergenetzwerk für 8 Elemente erforderlich. Inoue, Moriyama et al. (2007) benutzen Odd-Even Merge während, Chhugani et al. (2008) mit Unterstützung von Bitonic Merge die Mergeoperation implementieren. In Abb. 5.1 auf der nächsten Seite ist der Merge von Array `a` bestehend aus 3 Vektoren mit Array `b` bestehend aus 2 Vektoren dargestellt. Ein Vektor umfasst 4 Elemente. Zum Mergen werden die zwei Hilfsvektoren `v_min` und `v_max` verwendet. Der Merge, der insgesamt 5 Vektoren, läuft wie folgt ab:

(1) `v_min` mit erstem Vektor aus `a` initialisieren; `v_max` mit erstem Vektor aus `b` initialisieren; Indices `idx_a` und `idx_b` updaten; `v_min` und `v_max` mit einem Netzwerk mergen; nach dem Merge sind die kleinsten vier Elemente sortiert in `v_min`, die größten vier der acht Elementen sind sortiert in `v_max`; `v_min` in `c`

[3] Für kleinere n können außer Sortiernetzwerken auch andere Sortieralgorithmen benutzt werden. Inoue, Moriyama et al. (2007), die Erfinder des vektorisierten Mergesort, verwenden Combsort. Chhugani et al. (2008) benutzen dagegen für kleine n Sortiernetzwerke.

abspeichern; da 10 kleiner als 11 ist (`a[idx_a] < b[idx_b]`), nächsten Vektor aus `a` nach `v_min` laden und `idx_a` updaten

(2) `v_min` und `v_max` mit einem Netzwerk mergen; `v_min` in `c` abspeichern; da `a[idx_a] < b[idx_b]` nicht gilt, denn 24 ist größer als 11, nächsten Vektor aus `b` in `v_min` laden und `idx_b` updaten

(3) `v_min` und `v_max` mit einem Netzwerk mergen; `v_min` in `c` abspeichern; da Array `b` abgearbeitet ist, nächsten Vektor aus `a` in `v_min` laden und `idx_a` updaten

(4) `v_min` und `v_max` mit einem Netzwerk mergen; da beide Arrays abgearbeitet sind, `v_min` und `v_max` in `c` abspeichern

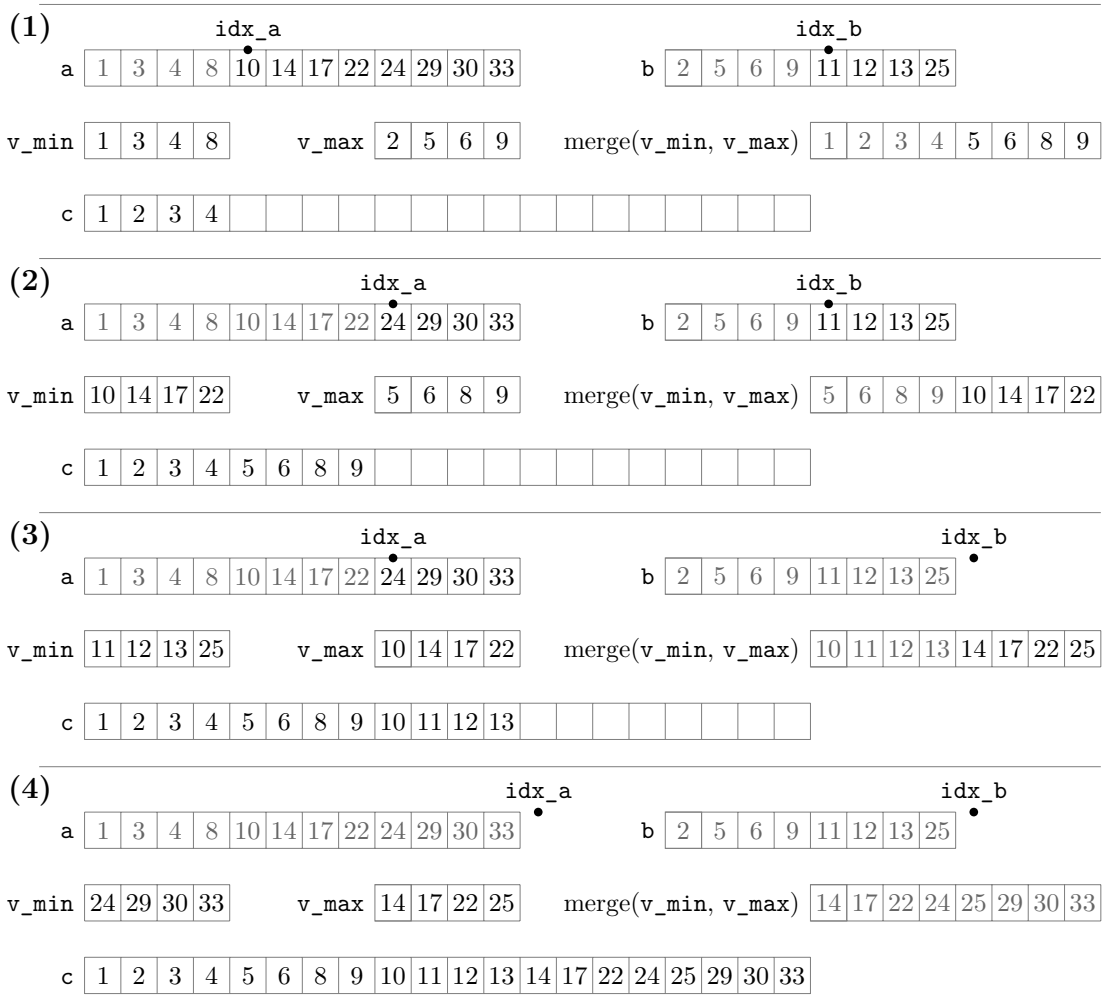


Abb. 5.1: Vektorisierte Mergeoperation

In jeder Iteration wird ein verzweigungsfreier Merge von zwei Vektorregistern mit einem vektorisierten Sortiernetzwerk durchgeführt. Die vektorisierte Mergeoperation

in Mergesort verringert im Vergleich zur skalaren Mergeoperation die Anzahl an bedingten Verzweigungen zur Ermittlung des nächsten Elements für das Ausgabearray `c`, da mit einer Iteration gleichzeitig vier Ausgabeelemente in `v_min` erzeugt werden. Dies reduziert die Anzahl an fehlerhaften Verzweigungsvorhersagen in modernen Prozessoren (vgl. Inoue und Taura 2015, S. 1276).

Die Funktionsweise der Mergeoperation in Mergesort ist für breitere SIMD-Vektoren äquivalent. Für Vektoren mit 8 Elementen wird ein Mergenetzwerk für mindestens 16 Elemente benötigt. Der Merge von zwei 256-Bit-Vektoren mit jeweils 8 sortierten Integern kann mit der Makro-Funktion `MERGE_16` aus Quelltext 4.3 auf Seite 30 durchgeführt werden. Acht einzelne Elemente können mit der Makro-Funktion `SORT_8` aus Quelltext 4.2 auf Seite 29 sortiert werden. Mit diesen beiden Makro-Funktionen und der intrinsischen Funktion `_mm256_extract_epi32`, die zum Extrahieren des ersten Elements eines Vektors dient, wird der skalare Mergesort aus Quelltext 5.1 auf Seite 42 zu einem vektorisierten Mergesort im Quelltext 5.2 transformiert.

Quelltext 5.2: Vektorisierter Mergesort

```
1  /* Arrays a und b mit a_size bzw. b_size Vektoren nach Array c mergen */
2  inline void merge_vectorized(__m256i *a, __m256i *b, __m256i *c,
3                               const int a_size, const int b_size, const int c_size) {
4    auto v_min = a[0], v_max = b[0];     /* v_min und v_max initialisieren */
5    int idx_a = 1, idx_b = 1;
6    for (int i = 0; i < c_size - 2; ++i) {
7      MERGE_16(v_min, v_max); /* sortierte Vektoren v_min und v_max mergen */
8      c[i] = v_min;                                /* Abspeichern von v_min */
9      if (idx_a == a_size) { v_min = b[idx_b++]; }       /* a abgearbeitet */
10     else if (idx_b == b_size) { v_min = a[idx_a++]; }  /* b abgearbeitet */
11     else { v_min = _mm256_extract_epi32(a[idx_a], 0) <
12                   _mm256_extract_epi32(b[idx_b], 0)
13                   ? a[idx_a++] :  b[idx_b++]; }}
14   MERGE_16(v_min, v_max); c[c_size - 2] = v_min; c[c_size - 1] = v_max; }
15
16 /* N Vektoren sortieren, c ist zusätzlicher Speicher der Länge N */
17 inline void merge_sort_vectorized(__m256i *vecs, __m256i *c, const int N) {
18   if (N == 1) { SORT_8(vecs[0]); return; } /* einzelnen Vektor sortieren */
19   if(N > 1) {
20     const int size_a = N / 2;                        /* Länge Teilfolge a */
21     const int size_b = N - size_a;                   /* Länge Teilfolge b */
22     merge_sort_vectorized(vecs, c, size_a);      /* Teilfolgen sortieren */
23     merge_sort_vectorized(vecs + size_a, c + size_a, size_b);
24     merge_vectorized(vecs, vecs + size_a, c, size_a, size_b, N); // mergen
25     for (int i = 0; i < N; ++i) vecs[i] = c[i]; }} /* kopieren nach vecs */
```

Die Rekursion endet in Zeile 18, sobald ein einzelner Vektor vorliegt. Dieser wird mit `SORT_8` sortiert. Die For-Schleife, die den Merge der zwei sortierten Arrays in Zeile 6 ausführt, bricht für die letzten beiden Vektoren ab. Diese werden anschließend in Zeile 14 gemergt und ins Ausgabearray `c` gespeichert. Statt mit einzelnen Zahlen rechnet der vektorisierte Mergesort aus Quelltext 5.2 mit Vektoren. Dies erfordert sowohl

ausgerichteten Speicher für das zu sortierende Array als auch für das Hilfsarray `c`. Ergänzend muss das Array ohne Rest durch 8 teilbar sein, damit der Algorithmus das komplette Array sortiert.[4]

5.3 Performanz des vektorisierten Mergesort

In Abb. 5.2 ist der Speedup der folgenden zwei Implementierungen des vektorisierten Mergesort ggü. `std::sort` dargestellt:

Mergesort naiv: Mergesort ohne Optimierungen aus Quelltext 5.2 auf Seite 44

Mergesort optimiert: (1) wenn Anzahl der Vektoren $N < 1025$, dann mit schritt- und modulbasiertem optimiertem Sortiernetzwerk aus Abschnitt 4.4.3 auf Seite 39 sortieren, (2) `v_min` und `v_max` bestehen jeweils aus vier Vektoren, die mit Bitonic Merge für 64 Elemente gemergt werden, (3) kein Kopieren der Werte aus Hilfsarray `c` nach `vecs`[5]

Ergänzend ist in Abb. 5.2 der Speedup von IPP-Radixsort[6] ggü. `std::sort` enthalten.

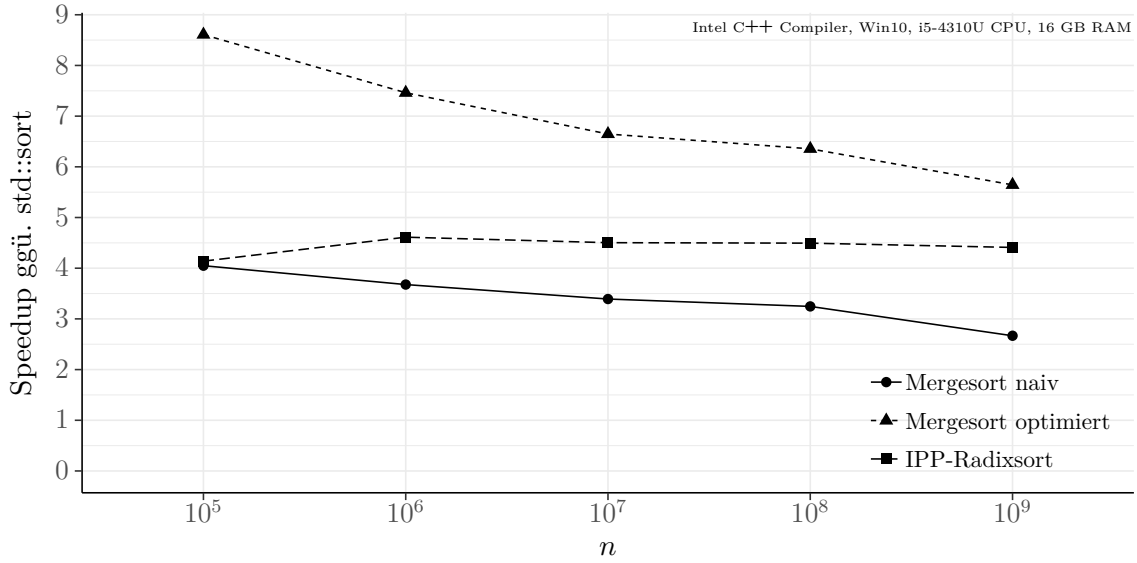


Abb. 5.2: Speedup des vektorisierten Mergesort ggü. std::sort

Die zu sortierenden Arrays sind mit gleichverteilten Zufallszahlen befüllt und liegen an 256-Bit-Grenzen ausgerichtet im Speicher vor. Ebenfalls ist das Hilfsarray der

[4] Mit Lade- und Speichern-Befehlen für nicht ausgerichteten Speicher bzw. der Verwendung von maskenbasierten Lade- und Speichern-Instruktionen, kann der Algorithmus für beliebige n und nicht ausgerichteten Speicher angepasst werden.

[5] Der optimierte vektorisierte Mergesort ist in Quelltext A.9 auf Seite 84 enthalten.

[6] Der verwendete Radixsort ist `ippsSortRadixAscend_32s_I_L`. Dieser benötigt wie die zwei vektorisierten Mergesort Varianten zusätzlichen Speicher der Größe n.

Länge n im Speicher an 256-Bit-Grenzen ausgerichtet. Die fünf Messpunkte für die Beschleunigungsfaktoren sind Zehnerpotenzen von $n = 10^5$ bis $n = 10^9$.

Bedingt durch die optimierten Sortiernetzwerke und das Caching bereits gemergter Arrays hat der optimierte Mergesort bei 100 000 Integern einen Beschleunigungsfaktor, der die Anzahl an Elementen innerhalb eines Vektorregisters übersteigt. Der Beschleunigungsfaktor nimmt jedoch mit wachsenden n ab und beträgt etwa 5,5 für $n = 1\,000\,000\,000$. Intels IPP-Radixsort hat dagegen ab ca. 1 000 000 Integer einen ungefähr konstanten Speedup ggü. `std::sort` von 4,5.

Der abnehmende Beschleunigungsfaktor resultiert aus der verminderten Bandbreite, mit der Daten aus langen Arrays für den Merge in die Vektorregister geladen werden. Kleinere Arrays passen beim Merge in den Cache und die Laufzeit hängt im Wesentlichen von der CPU ab. Ebenfalls können sich kleinere Arrays der vorherigen Mergeoperationen noch im Cache befinden, sodass sie beim erneuten Merge nicht aus dem Hauptspeicher geladen werden müssen. Der Speedup der Mergeoperation für große Arrays hängt dagegen von der Speicherbandbreite (*memory bandwidth*) ab, d. h. der Menge an Bytes, die pro Zeiteinheit aus dem Hauptspeicher geladen werden können. Die Speicherbandbreite verglichen mit der Anzahl an Bytes, die in derselben Zeiteinheit aus den Cache geladen werden können, ist gering. Es entstehen Verzögerungen während der Berechnung innerhalb der CPU.

Eine Strategie, den sinkenden Speedup zu vermeiden, ist statt zwei Arrays simultan zu mergen (*2-way merge*), mehrere Arrays gleichzeitig zu mergen, was als Multiway Mergesort bezeichnet wird. Chhugani et al. (2008) und Inoue und Taura (2015) vektorisieren Multiway Mergesort und können damit einen konstanten Speedup von etwa 3,3 für 128-Bit-Register mit vier Integern erzielen.[7]

Der vektorisierte Mergesort benötigt zusätzlichen Speicher der Größe n, was insbesondere für große Arrays bei nicht ausreichend Arbeitsspeicher problematisch sein kann. Im nächsten Kapitel wird Quicksort vektorisiert. Der vektorisierte Quicksort benötigt keinen linear wachsenden zusätzlichen Speicher und überbietet den Speedup des vektorisierten Mergesort.

[7] Chhugani et al. vergleichen den Speedup mit einer skalaren Implementierung ihres Algorithmus, während Inoue und Taura Strukturen sortieren und den Speedup mit `std::stable_sort` evaluieren.

6 Quicksort

6.1 Besonderheiten von Quicksort

Quicksort zählt zu den zehn einflussreichsten Algorithmen in der Entwicklung von Wissenschaft und Technik des 20. Jahrhunderts (Sullivan und Dongarra 2000).

In Quicksort wird das Array in zwei Teilarrays rekursiv zerlegt. Vor jeder erneuten Zerlegung des linken bzw. rechten Teilarrays werden die Elemente anhand eines Pivot-Werts dermaßen angeordnet, dass im linken Array alle Werte kleiner-gleich dem Pivot-Wert und im rechten Array größer als der Pivot-Wert sind. Der Zerlegungsvorgang des Arrays in zwei Teilarrays wird als Partitionierung bezeichnet. Der Pivot-Wert wird nach der Partitionierung zwischen dem linken und rechten Teilarray an seine korrekte Position im sortierten Array gesetzt. Im ungünstigsten Fall ist der Pivot-Wert das Minimum oder das Maximum des Arrays. In diesem Fall hat ein Subarray nach der Teilung die Länge 0, während das andere Subarray die restlichen Elemente ohne dem Pivot-Wert beinhaltet. Tritt bei der Partitionierung immer der ungünstigste Fall ein, dann hat Quicksort eine Laufzeit von $\mathcal{O}(n^2)$.[1] Durch zufälliges Mischen der Elemente vor der Sortierung ist der Worst-Case auch bei einer deterministischen Pivot-Strategie unwahrscheinlich.[2]

Im Mittel hat Quicksort für die Anzahl der Vergleiche eine Komplexität wie Mergesort von $\mathcal{O}(n \log n)$ (siehe Herold 2017, S. 503). Die tatsächliche durchschnittliche Anzahl an Vergleichen ist bei Quicksort um 39 Prozent höher als bei Mergesort. Normalerweise ist Quicksort trotz der zusätzlichen Vergleiche schneller als Mergesort, da weniger Datenbewegungen stattfinden. Die innere Schleife vergleicht während der Partitionierung alle Arrayelemente mit einem festen Wert, der im Register gehalten werden kann. Eine Vertauschung (Datenbewegung) findet erst statt, wenn die Arrayelemente im falschen Teilarray liegen (vgl. Sedgewick 2014, S. 318 ff.). Beim Mergesort dagegen werden alle Elemente sukzessiv ins Hilfsarray bewegt. Vergli-

[1] Quelltext A.10 auf Seite 85 enthält eine naive Implementierung von Quicksort ohne eine Strategie zur Vermeidung des Worst-Case.

[2] Die Wahrscheinlichkeit, dass nach zufälligem Mischen die Laufzeit zur Sortierung eines Arrays mit 1 000 000 Elementen den Durchschnittswert um mehr als das Zehnfache übersteigt, ist der Tschebyschow-Ungleichung zufolge geringer als 0,001 Prozent (vgl. Sedgewick 2014, S. 320).

chen wird bei Mergesort nicht mit einem festen Wert, sondern mit einem anderen Arrayelement.

Wird Quicksort *in-place* ohne zusätlichen Speicher implementiert, dann ist die Sortierung instabil und die relative Reihenfolge der Datensätze mit gleichem Schlüssel kann sich ändern.[3] Der beste Pivot-Wert für die Partitionierung des Arrays ist der Median, denn er teilt das Array in zwei balancierte Teilarrays auf. Eine Berechnung des Medians des kompletten Arrays würde die Ausführungszeit verlangsamen. Pivot-Strategien, in denen der Median aus einer ungeraden Anzahl an zufällig gezogenen oder deterministischen Werten des Arrays ermittelt wird (z. B. Median-aus-drei), können dagegen die Ausführungszeit von Quicksort verbessern (vgl. Sedgewick 2014, S. 321).

Enthält das Array eine große Anzahl doppelter Werte, dann degradiert die Performanz von Quicksort. Im Worst-Case sind alle Arrayelemente gleich und die Laufzeit beträgt unabhängig von der Pivot-Strategie $\mathcal{O}(n^2)$. Wird das Array statt in zwei Abschnitte in drei Abschnitte mit Elementen „kleiner als“, „gleich“ und „größer als“ der Pivot-Wert geteilt, dann sind alle Elemente, die „gleich“ dem Pivot-Wert sind, nach der Partitionierung sortiert. Falls die Anzahl gleicher Werte im Array nicht hoch ist, benötigt Quicksort mit 3-Wege-Partitionierung mehr Vergleichs- bzw. Tauschoperationen als Quicksort mit 2-Wege-Partitionierung.[4]

6.2 Quicksort ohne quadratischen Worst-Case

Daoud et al. (2011) implementieren eine Pivot-Strategie basierend auf Median-aus-drei und Binary-Quicksort[5]. Im Worst-Case hat Quicksort mit dieser Strategie für die Sortierung von 32-Bit-Integern eine Zeitkomplexität von $2 \cdot \mathcal{O}(32 \cdot n)$.[6] Der Algorithmus startet die Partitionierung des Arrays mit Median-aus-drei. Falls durch den Median-aus-drei zwei unbalancierte Teilarrays entstehen, wird im linken Teilarray der Mittelwert zwischen -2^{31} und aktuellem Pivot-Wert, im rechten Teilarray der Mittelwert zwischen aktuellem Pivot-Wert + 1 und $2^{31} - 1$ im nächsten Quicksort-Aufruf verwendet. Der Pivot-Wert muss nicht im Array enthalten sein. Möglich ist,

[3] Auf die Sortierung von einfachen Datentypen wie Integer hat die Stabilität eines Sortieralgorithmus keinen Einfluss. Der vektorisierte Mergesort im Quelltext 5.2 auf Seite 44 ist ebenfalls instabil, da Sortiernetzwerke die Mergeoperation ausführen.

[4] In der Literatur wird die 3-Wege-Partitionierung als das *Dutch National Flag Problem* bezeichnet. Bentley und McIlroy (1993) optimieren Quicksort mit 3-Wege-Partitionierung.

[5] Binary-Quicksort wird auch als Radix-Exchange bezeichnet und ist ein *in-place* Radixsort mit zwei *buckets* (siehe Nebel 2012, S. 233 f.).

[6] Die Zeitkomplexität im Worst-Case für die Sortierung von 64-Bit-Integern wäre $2 \cdot \mathcal{O}(64 \cdot n)$. Mit der Pivot-Strategie von Daoud et al. (2011) beträgt die Worst-Case Komplexität von Quicksort das doppelte der Worst-Case Komplexität von Binary-Quicksort.

dass alle Elemente im Array größer bzw. kleiner als der Pivot-Wert sind. Wenn die resultierenden Teilarrays unbalanciert sind, dann wird in der darauffolgenden Partitionierung wieder der Median-aus-drei verwendet, ansonsten wird als Pivot erneut der Durchschnitt zwischen kleinstmöglichen und höchstmöglichen Wert in einem Teilarray verwendet. Im linken Teilarray ist der höchstmögliche Wert der aktuelle Pivot-Wert und der kleinstmögliche ist aus vorhergehenden Quicksort-Aufrufen bekannt. Im rechten Teilarray ist der kleinstmögliche Wert der aktuelle Pivot-Wert + 1 und der höchstmögliche ist aus vorhergehenden Quicksort-Aufrufen bekannt. Sobald eine Pivot-Berechnung zu unbalancierten (entarteten) Teilarrays führt, wird in der nächsten Rekursionsebene die andere Pivot-Berechnung verwendet.

Daoud et al. implementieren ihren Quicksort mit 2-Wege-Partitionierung in Java und bewerten die Performanz mit Zufallszahlen befüllten Arrays. Sie ermitteln den Speedup gegenüber Quicksort von Bentley und McIlroy (1993) mit 3-Wege-Partitionierung, welcher effizient doppelte Werte sortieren kann. Ein Vergleich der Laufzeiten für Fälle, in denen das Array eine große Anzahl doppelter Werte enthält, fehlt bei Daoud et al. Es ist nicht ersichtlich, wie Daoud et al. wiederholende Werte im Array behandeln.[7]

Aufbauend auf der Arbeit von Daoud et al. (2011) wird im Quelltext 6.1 auf der nächsten Seite ein *in-place* Quicksort mit 2-Wege-Partitionierung vorgestellt, der sowohl den quadratischen Worst-Case vermeidet als auch Teilarrays mit gleichen Werten erkennt. Die skalare Implementierung des Algorithmus benötigt für die Sortierung von Arrays, die mit gleichverteilten Zufallszahlen befüllt sind, in etwa 80 % mehr Zeit als `std::sort`. Als skalarer Sortieralgorithmus kann er nicht mit anderen optimierten Sortieralgorithmen konkurrieren. Vektorisiert in Kombination mit Sortiernetzwerken für kleine n und ergänzenden Optimierungen ist der Algorithmus dagegen der schnellste der in dieser Arbeit betrachteten vektorisierten Sortieralgorithmen.

Die Quicksort-Funktion `qs_core` in Zeile 12 hat zwei ergänzende Parameter. Der Parameter `choose_avg` wird auf `true` gesetzt, falls als Pivot-Wert `avg`, d. h. der Durchschnitt vom kleinsten und größten Wert im Array, zu wählen ist. Ansonsten wird in Zeile 16 der Wert in der Mitte des Arrays als Pivot gewählt. Während der Partitionierung wird der kleinste und der größte Wert im Array in Zeile 7 ermittelt und in den per Referenz übergebenen Variablen `smallest` und `biggest` gespeichert. In Zeile 23 wird überprüft, ob die aktuelle Pivot-Strategie balancierte Arrays produziert. Sind beide Teilarrays unbalanciert, dann wird für den nächsten

[7] In der Praxis kommen Arrays mit einer großen Anzahl doppelter Schlüssel laut Sedgewick (2014, S. 321) häufig vor. Als Beispiele führt Sedgewick die Sortierung nach Geburtsjahr oder nach Geschlecht auf.

Rekursionsaufruf die Pivot-Strategie verändert. Die Schwelle von `0.2` zur Festlegung einer entarteten Partitionierung stammt aus dem von Daoud et al. (2011) empfohlenen Bereich zwischen 0,001 und 0,3. Die Rekursion wird für ein Teilarray beendet, wenn die Werte darin alle gleich sind. Im linken Teilarray sind alle Werte gleich, falls der aktuelle Pivot-Wert dem kleinsten Wert im Array entspricht (vgl. Zeile 24). Im rechten Teilarray sind alle Werte gleich, wenn der Pivot-Wert + 1 identisch mit dem größten Wert des Arrays ist (vgl. Zeile 26). Zur Berechnung der Durchschnittswerte für die beiden Teilarrays in den Zeilen 25 und 27 wird die Funktion `average` aus Zeile 2 verwendet.[8]

Quelltext 6.1: Nichtquadratischer Quicksort

```
1  /* Durchschnitt von zwei Integern ohne Überlauf */
2  inline int average(int a, int b) { return (a & b) + ((a ^ b) / 2); }
3
4  inline int partition(int *arr, int left, int right,   /* Partitionierung */
5                       int pivot, int &smallest, int &biggest) {
6    while(left < right) {          /* smallest und biggest sind Referenzen */
7      smallest = min(smallest, arr[left]); biggest = max(biggest, arr[left]);
8      if (arr[left] > pivot) { swap(arr[left], arr[--right]); }
9      else { ++left; }}
10   return left; }
11
12 inline void qs_core(int *arr, int left, int right,
13                      bool choose_avg = false, const int avg = 0) {
14   if (left < right) {                 /* falls mind. 2 Elemente im Array */
15     /* avg ist Durchschnitt vom größten und kleinsten Wert im Array */
16     int pivot = choose_avg ? avg : arr[average(left, right)];
17     int smallest = INT32_MAX;     /* kleinster Wert nach Partitionierung */
18     int biggest = INT32_MIN;        /* größter Wert nach Partitionierung */
19     int bound = partition(arr, left, right + 1, pivot, smallest, biggest);
20     /* Anteil der kleineren Partition am Array */
21     double ratio = (min(right-(bound-1),bound-left)/double(right-left+1));
22     /* falls unbalancierte Teilarrays, Pivot-Berechnung ändern */
23     if (ratio < 0.2) { choose_avg = !choose_avg; }
24     if (pivot != smallest)      /* Werte im linken Teilarray verschieden */
25       qs_core(arr, left, bound - 1, choose_avg, average(smallest, pivot));
26     if (pivot + 1 != biggest)  /* Werte im rechten Teilarray verschieden */
27       qs_core(arr, bound, right, choose_avg, average(biggest, pivot)); }}
28
29 inline void qs(int *arr, int n) { qs_core(arr, 0, n - 1); }
```

Der vorgestellte Algorithmus führt unnötige Berechnungen in Zeile 7 aus, wenn die Partitionierung mit dem mittigsten Element des Arrays balancierte Teilarrays produziert und dadurch `avg` im nächsten Rekursionsaufruf nicht als Pivot verwendet wird. Ferner ist die Berechnung von dem kleinsten und größten Wert in Zeile 7

[8] Die Funktion `average` ist aus Dietz (2010) entnommen. Es wird immer abgerundet, d. h. `average(-2, -1)` ist –2. Ohne das Abrunden in `average` würde der Algorithmus aus Quelltext 6.1 nicht funktionieren.

lediglich in der ersten Partitionierung des Gesamtarrays nicht redundant, da in den nachfolgenden Partitionierungen für das linke Teilarray der aktuelle kleinste Wert ebenfalls der kleinste Wert ist und für das rechte Teilarray der aktuelle größte Wert auch der größte Wert ist. Darüber hinaus finden während der Partitionierung in Zeile 8 unnötige Vertauschungen (Datenbewegungen) statt. Ist der aktuelle Wert `arr[left]` größer als der Pivot-Wert, wird er mit einem unbekannten Wert auf der rechten Seite des Arrays vertauscht. Falls der unbekannte Wert aber größer als der Pivot-Wert ist, dann wird er im nächsten Iterationsschritt wieder zurück auf die rechte Seite des Arrays vertauscht. Der Algorithmus führt somit unnötige und redundante Berechnungen bzw. Vertauschungen aus, was die relativ schlechte Performanz im Vergleich zu `std::sort` bewirkt.[9]

Die Worst-Case Laufzeit für den nichtquadratischen Quicksort aus Quelltext 6.1 auf Seite 50 beträgt wie beim Quicksort von Daoud et al. $2 \cdot \mathcal{O}(32 \cdot n)$, jedoch ist der kleinste bzw. größte Wert des zu partitionierenden Arrays bekannt, sodass bei Verwendung der Durchschnitts-Pivots erwartungsgemäß weniger Rekursionsaufrufe notwendig sind, um das komplette Array zu sortieren.

Der gewöhnliche Quicksort wie im Quelltext A.10 auf Seite 85 dargestellt, setzt den Pivot-Wert nach der Partitionierung an seine sortierte Stelle im Gesamtarray. Der nichtquadratische Quicksort partitioniert das Array, ohne einen Wert an seine sortierte Position zu setzen. Dies ermöglicht, im nichtquadratischen Quicksort Werte als Pivot zu verwenden, die im Array nicht vorhanden sind. Auch ist beim nichtquadratischen Quicksort die Ermittlung des Medians aus mehreren Werten des Arrays als Pivot-Strategie effizienter, da der Index des Pivot-Werts nicht benötigt wird. Im gewöhnlichen Quicksort wird die Rekursion beendet, sobald das Teilarray weniger als zwei Elemente enthält. Der nichtquadratische Quicksort beendet die Rekursion dagegen, sobald alle Elemente im Teilarray gleich sind.[10]

Die Partitionierung in Zeile 8 und 9 ermöglicht es, das Array zu kürzen. In jedem Iterationsschritt wird ein Wert auf die korrekte Seite partitioniert, wobei das zu partitionierende Array um ein Element abnimmt. Für die vektorisierte Partitionierung werden ohne Rest durch 8 teilbare Arraylängen benutzt. Mit der skalaren Partitionierung kann das Array auf die zur Vektorisierung erforderliche Länge gekürzt werden.

[9] Auf Optimierungen wurde verzichtet, um eine kompakte Darstellung des Algorithmus zu gewährleisten.

[10] Ohne die Überprüfung auf gleiche Werte würde der Algorithmus aus Quelltext 6.1 auf Seite 50 für doppelte Werte nicht terminieren, da während der Partitionierung der Pivot-Wert nicht an seine sortierte Stelle im Gesamtarray gesetzt wird.

6.3 Vektorisierung der Partitionierungsfunktion

6.3.1 Schematischer Ablauf der Partitionierung

Eine Vorgehensweise zur effizienten Vektorisierung von Quicksort auf modernen Architekturen mit SIMD Vektorregistern war bis zur Arbeit von Gueron und Krasnov (2016) nicht bekannt. Chhugani et al. (2008, S. 1314), die einen vektorisierten Multiway Mergesort implementieren, schreiben über die Vektorisierung von Quicksort das Folgende:

> *Quicksort has been one of the fastest algorithms used in practice. However, its efficient implementation for exploiting SIMD is not known.*

Inoue und Taura (2015, S. 1274), die ebenfalls einen vektorisierten Multiway Mergesort erstellen, äußern ihren Standpunkt hinsichtlich der Vektorisierung von Quicksort folgendermaßen:

> *By using the SIMD instructions efficiently in the merge operation, multiway mergesort outperforms other comparison-based sorting algorithms, such as quicksort, that are not suitable for exploiting the SIMD instructions.*

Dass Quicksort geeignet (*suitable*) ist, um mit Vektorinstruktionen implementiert zu werden, zeigen die Arbeiten von Gueron und Krasnov (2016) bzw. Bramas (2017). Während Gueron und Krasnov (2016) den Quicksort mit zusätzlichem Speicher vektorisieren, gelingt es Bramas (2017), die Partitionierung ohne linear wachsenden zusätzlichen Speicher zu realisieren. Die von Gueron und Krasnov bzw. Bramas erstellten Quicksort-Algorithmen haben jedoch dieselben Schwächen wie der skalare naiv implementierte Quicksort aus Quelltext A.10 auf Seite 85.[11]

Da AVX2 keine Komprimierungsbefehle hat, benutzen Gueron und Krasnov (2016) in einer Lookup-Tabelle gespeicherte Shuffle-Masken, um einen einzelnen Vektor entsprechend dem Pivot-Wert zu partitionieren. Pro Vektor wird zweimal geshuffelt. Der erste Shuffle ordnet die Elemente, die kleiner-gleich dem Pivot-Wert sind, in einem Vektorregister sequentiell an. Der zweite Shuffle ordnet die Elemente, die größer als der Pivot-Wert sind, in einem zweiten Vektorregister sequentiell an. Es wird ein Hilfsarrays zum Abspeichern der Vektorregister für Werte, die kleiner-gleich Pivot-Wert und ein Hilfsarray für Werte, die größer als der Pivot-Wert sind, verwendet. Am Ende der Partitionierung werden beide Hilfsarrays konkateniert.

[11] Die zwei größten Schwachpunkte sind: Worst-Case $\mathcal{O}(n^2)$ und keine Strategie für Arrays mit hoher Anzahl doppelter Werte.

In der *in-place* Partitionierung von Bramas (2017) werden dagegen die zwei an den äußeren Enden des Arrays liegenden SIMD-Vektoren zu Beginn der Partitionierung zwischengespeichert. Dadurch entsteht am Anfang und am Ende des Arrays Platz zum Speichern der Elemente während der Partitionierung. Da Bramas den Quicksort in AVX-512 implementiert, werden zur Partitionierung eines Vektors und dessen Speicherung zwei Komprimierungs-Speichern-Befehle angewendet.

Der vektorisierte Partitionierungs-Algorithmus in dieser Arbeit ist ein Hybrid aus der Strategie von Gueron und Krasnov (2016) und Bramas (2017) mit zusätzlichen Optimierungen. Die Vorgehensweise zur Partitionierung eines Arrays wird in Abb. 6.1 anhand von Vektoren, die vier Elemente aufnehmen können, demonstriert.

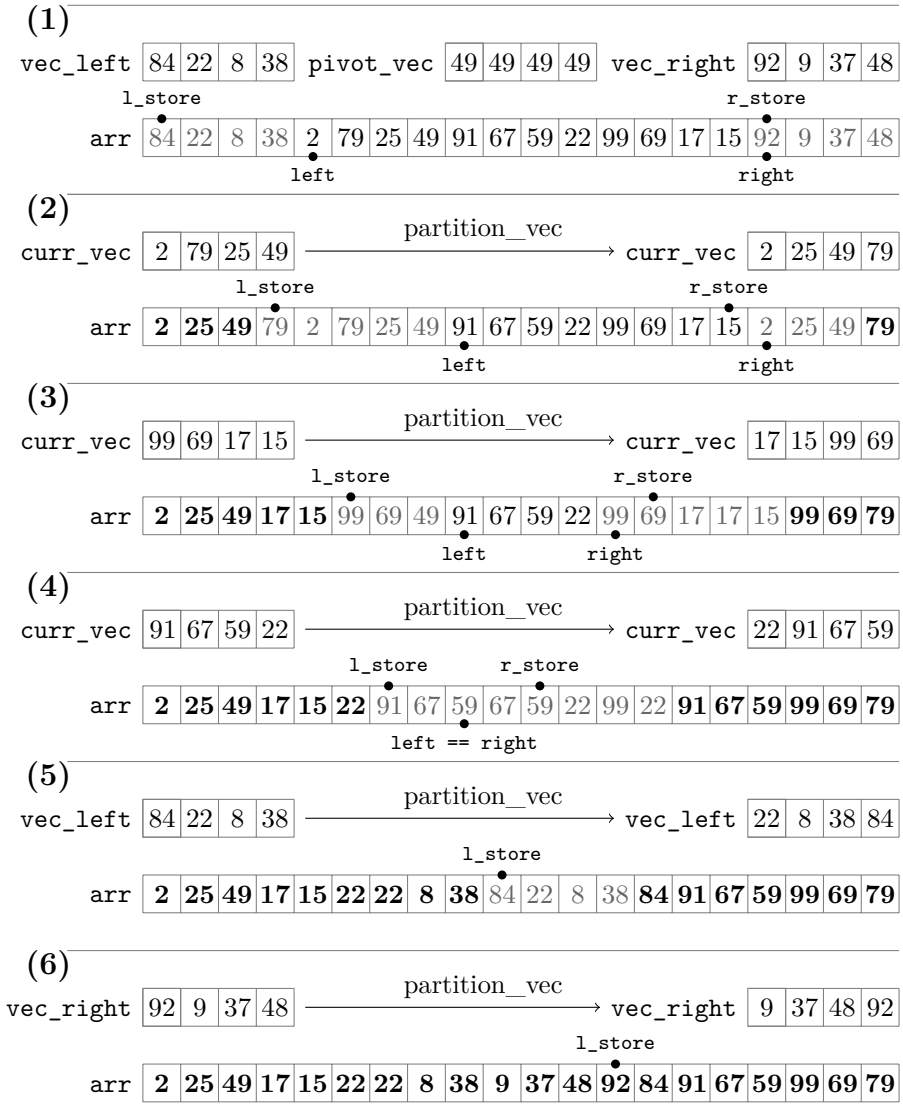


Abb. 6.1: Schematischer Ablauf der vektorisierten Partitionierung des Arrays

Im Beispiel besteht das zu partitionierende Array `arr` aus 20 Elementen. Der Pivot-Wert ist 49. Die Partitionierung eines einzelnen Vektors wird mit der Funktion

`partition_vec` durchgeführt. Ein Pfeil in Abb. 6.1 auf Seite 53 kennzeichnet die Partitionierung eines einzelnen Vektors anhand des Pivot-Vektors `pivot_vec`. In der Funktion `partition_vec` werden die Elemente kleiner-gleich dem Pivot-Wert am Anfang des Vektors und die Elemente größer als der Pivot-Wert, danach im Vektor angeordnet. Die Indices `l_store` und `r_store` kennzeichnen die Speicherstellen im Array für einen partitionierten Vektor. Die Indices `left` und `right` werden dagegen benutzt, um die Positionen der zu verarbeitenden Vektoren zu ermitteln. Der detaillierte Ablauf in Abb. 6.1 auf Seite 53 ist wie folgt:

(1) Zu Beginn der Partitionierung werden die ersten vier Elemente des Arrays `arr` im Vektor `vec_left` und die letzten vier Elemente im Vektor `vec_right` gespeichert. Die Vektoren `vec_left` und `vec_right` werden erst, nachdem alle anderen Vektoren abgearbeitet sind, partitioniert.

(2) Der Vektor unter Index `left` wird nach `curr_vec` geladen, partitioniert und an den äußeren Enden des Arrays gespeichert. Der Speicherpunkt `l_store` verschiebt sich um drei Index-Stellen nach rechts, da in `curr_vec` drei Elemente kleiner-gleich dem Pivot-Wert sind. Äquivalent verschiebt sich `r_store` um eine Index-Stelle nach links, da in `curr_vec` ein Element größer als der Pivot-Wert ist. Der zu partitionierende Vektor `curr_vec` stammt von der linken Seite des Arrays, deshalb wird der Index `left` um vier Index-Stellen nach rechts verschoben.

(3) Da `arr` rechts weniger partitionierte Elemente enthält als links (ein Element rechts und drei links), stammt der zu partitionierende Vektor `curr_vec` von der rechten Seite des Arrays `arr`. Nach der Partitionierung von `curr_vec` und seiner Abspeicherung auf beiden Seiten entsprechend den Speicherpunkten `l_store` und `r_store` sind links insgesamt fünf und rechts drei Elemente partitioniert. Die Speicherstellen `l_store` und `r_store` werden aktualisiert und `right` um vier Index-Stellen nach links verschoben.

(4) Der zu partitionierende Vektor `curr_vec` wird wieder anhand des Index `right` bestimmt, da die rechte Seite des Arrays `arr` weniger partitionierte Elemente enthält als die linke. Der Vektor `curr_vec` wird partitioniert und gemäß den Speicherpunkten `l_store` und `r_store` gespeichert. Die Indices und Speicherpunkte werden aktualisiert. Die Indices `left` und `right` sind gleich, d. h. die Vektoren `vec_left` und `vec_right` sind noch zu partitionieren, bevor das komplette Array partitioniert ist.

(5) Den anfangs gespeicherten Vektor `vec_left` partitionieren und auf beiden Seiten des Arrays entsprechend `l_store` und `r_store` abspeichern. Drei Elemente sind kleiner als der Pivot-Wert, deshalb wird `l_store` um drei Index-Stellen nach rechts verschoben.

(6) Den am Anfang gespeicherten Vektor `vec_right` partitionieren und gemäß `l_store` abspeichern. Da in `vec_right` drei Elemente kleiner als der Pivot-Wert sind, wird `l_store` um drei Index-Stellen nach rechts verschoben.

Der partitionierte Vektor wird links und rechts im Array gespeichert. Der nächste zu partitionierende Vektor wird von derjenigen Seite des Arrays mit weniger korrekt partitionierten Elementen entnommen. Ein Überschreiben noch nicht partitionierter Werte des Arrays ist unmöglich, da durch die beiden anfangs zwischengespeicherten Vektoren Platz auf jeder Seite zum Überschreiben eines Vektors besteht. Dieser Platz wird erst aufgebraucht, sobald die am Anfang zwischengespeicherten Vektoren partitioniert werden.

6.3.2 Partitionierung eines Vektors

Zur Partitionierung einzelner Vektoren werden ähnlich wie bei Gueron und Krasnov (2016) in einer Lookup-Tabelle gespeicherte Permutations-Masken benutzt. Mit dem Algorithmus von Gueron und Krasnov wären zwei Permutationen notwendig, um einen Vektor zu partitionieren, da die Werte kleiner-gleich bzw. größer als der Pivot-Wert von links nach rechts in das jeweilige Hilfsarray gespeichert werden. In der vorgestellten *in-place* Partitionierung aus Abb. 6.1 auf Seite 53 reicht eine Permutation aus, um einen Vektor zu partitionieren, da das Array für Werte, die kleiner-gleich dem Pivot-Wert sind, von links nach rechts und für Werte, die größer als der Pivot-Wert sind, von rechts nach links aufgefüllt wird. Zur Realisierung aller Möglichkeiten für die Partitionierung eines Vektors mit acht Elementen werden 256 Permutations-Masken benötigt.[12]

In Quelltext 6.2 auf der nächsten Seite ist die Partitionierung eines Vektors in AVX2 dargelegt. Die Funktion `partition_vec` erhält per Referenz den Vektor, der zu partitionieren ist (`curr_vec`), und den Pivot-Vektor (`pivot_vec`). Ebenfalls werden in der Funktion `partition_vec` die Vektoren mit den kleinsten (`smallest_vec`) und größten Werten (`biggest_vec`) aktualisiert, damit zum Schluss der Partitionierung des Arrays der kleinste bzw. größte Wert im Array ermittelt werden kann.

[12] Eine Zahl ist entweder kleiner-gleich oder größer als der Pivot-Wert. Für acht Zahlen ergeben sich $2^8 = 256$ Konstellationen.

Quelltext 6.2: Partitionierung eines Vektors

```
1  inline int partition_vec(__m256i &curr_vec, const __m256i &pivot_vec,
2                           __m256i &smallest_vec, __m256i &biggest_vec) {
3    /* welche Elemente sind größer als Pivot */
4    __m256i compared = _mm256_cmpgt_epi32(curr_vec, pivot_vec);
5    /* Vektoren mit kleinsten und größten Werten updaten */
6    smallest_vec = _mm256_min_epi32(curr_vec, smallest_vec);
7    biggest_vec = _mm256_max_epi32(curr_vec, biggest_vec);
8    /* höchstes Bit aus jedem Integer des Vektors extrahieren */
9    int mm = _mm256_movemask_ps(reinterpret_cast<__m256>(compared));
10   /* Anzahl der Einsen, jede 1 steht für Element größer als Pivot */
11   int amount_gt_pivot = _mm_popcnt_u32((mm));
12   /* Elemente größer als Pivot nach rechts, kleiner-gleich nach links */
13   curr_vec = _mm256_permutevar8x32_epi32(curr_vec, permutation_masks[mm]);
14   /* Anzahl der Elemente größer als Pivot zurückgeben */
15   return amount_gt_pivot; }
```

In Zeile 4 wird ein Vektor berechnet, der angibt, welche Elemente größer als der Pivot-Wert sind. Basierend auf diesem Vektor wird in Zeile 9 der Index für die benötigte Permutations-Maske ermittelt. Das Array `permutation_masks` enthält für jeden möglichen Index die passende Permutations-Maske.[13] Nach der Partitionierung von `curr_vec` mittels Permutation in Zeile 13 gibt die Funktion `partition_vec` die Anzahl an Elementen, die größer als der Pivot-Wert sind, zurück.

6.3.3 Partitionierung des Arrays

Die Partitionierung des Arrays wie schematisch in Abb. 6.1 auf Seite 53 gezeigt kann mit dem Algorithmus aus Quelltext 6.3 auf der nächsten Seite erfolgen. Das zu partitionierende Array wird in der For-Schleife von Zeile 3 mit skalarer Partitionierung gekürzt, sodass die Anzahl an Elementen darin ohne Rest durch 8 teilbar ist. In der While-Schleife aus Zeile 31 erfolgt die vektorisierte Partitionierung des Arrays. Das Vorgehen zur vektorisierten Partitionierung des Arrays mit Vektorregistern, die acht Elemente aufnehmen, ist ähnlich zur Partitionierung mit vierelementigen Vektorregistern aus Abb. 6.1 auf Seite 53. Während der Partitionierung eines einzelnen Vektors mit der Funktion `partition_vec` in Zeile 39 wird ergänzend der Vektor mit den kleinsten Werten (`sv`) und der Vektor mit den größten Werten (`bv`) des Arrays aktualisiert. Nach der While-Schleife werden die am Anfang zwischengespeicherten Vektoren `vec_left` und `vec_right` partitioniert. In Zeile 53 wird im Vektor mit den kleinsten Werten (`sv`) das kleinste und im Vektor mit den größten Werten (`bv`) das größte Element bestimmt.[14]

[13] Das Programm zur Erzeugung des Arrays mit Permutations-Masken ist in Quelltext A.11 auf Seite 85 enthalten.

[14] Die Implementierung von `calc_min` und `calc_max` ist im Quelltext A.12 auf Seite 86 enthalten.

Quelltext 6.3: Vektorisierte Partitionierung des Arrays

```
inline int partition_vectorized_8(int *arr, int left, int right,
                                  int pivot, int &smallest, int &biggest) {
  for (int i = (right - left) % 8; i > 0; --i) {         /* Array kürzen */
    smallest = min(smallest, arr[left]); biggest = max(biggest, arr[left]);
    if (arr[left] > pivot) { swap(arr[left], arr[--right]); }
    else { ++left; }}

  if(left == right) return left;      /* weniger als 8 Elemente im Array */

  auto pivot_vec = _mm256_set1_epi32(pivot); /* Vektor befüllt mit Pivot */
  auto sv = _mm256_set1_epi32(smallest);          /* Vektor für smallest */
  auto bv = _mm256_set1_epi32(biggest);            /* Vektor für biggest */

  if(right - left == 8) {         /* falls 8 Elemente übrig nach Kürzung */
    auto v = LOAD_VEC(arr + left);
    int amount_gt_pivot = partition_vec(v, pivot_vec, sv, bv);
    STORE_VEC(arr + left, v);
    smallest = calc_min(sv); biggest = calc_max(bv);
    return left + (8 - amount_gt_pivot); }

  /* erste und letzte 8 Werte werden zum Schluss partitioniert */
  auto vec_left = LOAD_VEC(arr + left);                /* erste 8 Werte */
  auto vec_right = LOAD_VEC(arr + (right - 8));       /* letzte 8 Werte */
  /* Positionen, an denen die Vektoren gespeichert werden */
  int l_store = left;                    /* linke Position zum Speichern */
  int r_store = right - 8;              /* rechte Position zum Speichern */
  /* Positionen, an denen Vektoren geladen werden */
  left += 8;        /* erhöhen, da erste 8 Elemente zwischengespeichert */
  right -= 8;   /* verringern, da letzte 8 Elemente zwischengespeichert */

  while(right - left != 0) {  /* 8 Elemente pro Iteration partitionieren */
    __m256i curr_vec;                   /* Vektor, der partitioniert wird
    */
    /* wenn weniger Elemente auf rechter Seite des Arrays gespeichert, dann
     * nächster Vektor von rechter Seite, sonst von linker Seite laden */
    if((r_store + 8) - right < left - l_store) {
      right -= 8; curr_vec = LOAD_VEC(arr + right); }
    else { curr_vec = LOAD_VEC(arr + left); left += 8; }
    /* aktuellen Vektor partitionieren und auf beiden Seiten speichern */
    int amount_gt_pivot = partition_vec(curr_vec, pivot_vec, sv, bv);
    STORE_VEC(arr + l_store, curr_vec); STORE_VEC(arr + r_store, curr_vec);
    /* Positionen, an denen die Vektoren gespeichert werden, updaten */
    r_store -= amount_gt_pivot; l_store += (8 - amount_gt_pivot); }

  /* vec_left partitionieren und speichern */
  int amount_gt_pivot = partition_vec(vec_left, pivot_vec, sv, bv);
  STORE_VEC(arr + l_store, vec_left); STORE_VEC(arr + r_store, vec_left);
  l_store += (8 - amount_gt_pivot);
  /* vec_right partitionieren und speichern */
  amount_gt_pivot = partition_vec(vec_right, pivot_vec, sv, bv);
  STORE_VEC(arr + l_store, vec_right);
  l_store += (8 - amount_gt_pivot);

  smallest = calc_min(sv); biggest = calc_max(bv);
  return l_store; }
```

Der Algorithmus kann durch die Simulation von breiteren Vektorregistern beschleunigt werden. In Quelltext A.13 auf Seite 86 ist ein vektorisierter Quicksort mit einer simulierten Registerbreite von 64 Elementen gezeigt. Zu Beginn der Partitionierung werden von beiden Seiten des Arrays jeweils 8 Vektoren zwischengespeichert. In jeder Iteration der While-Schleife werden 8 Vektoren partitioniert und auf beiden Seiten des Arrays gespeichert.[15]

Durch die Simulation von breiteren Vektorregistern kann die durchschnittliche Latenz von Vektorinstruktionen verringert werden. Beispielsweise hat eine am Speicher nicht ausgerichtete Lade-Instruktion eines Vektors eine Latenz von 1 Taktzyklus und einen Durchsatz von 0,25 Takte. Werden mehrere Lade-Instruktionen hintereinander ausgeführt, dann können voneinander unabhängige Lade-Instruktionen mit einer Zeitverzögerung von 0,25 Taktzyklen beginnen. Bei Ausführung einer einzigen Lade-Instruktion pro Iteration würde dagegen das Rechenwerk, das für die Lade-Instruktion zuständig ist, 0,75 Taktzyklen untätig sein. Ferner wird durch das sequentielle Laden von mehreren hintereinanderliegenden Vektoren die Cache-Ausnutzung verbessert, was die Ausführungszeit zusätzlich reduziert.

Ebenfalls kann die sequentielle bzw. zeitgleiche Partitionierung von mehreren Vektoren die durchschnittliche Latenz der darin benötigten Vektorinstruktionen senken. Zum Beispiel die Vergleichs-Instruktion `_mm256_cmpgt_epi32` in Quelltext 6.2 auf Seite 56, welche die Elemente in einem Vektor, die größer als der Pivot-Wert sind, ermittelt, hat auf Haswell-Prozessoren eine Latenz von 5 Taktzyklen, wobei der Durchsatz, d. h. die physische Beanspruchung des zuständigen Rechenwerks, 1 Taktzyklus beträgt.

6.4 Vektorisierung der Pivot-Strategie

6.4.1 Median der Mediane als Pivot-Strategie

In den folgenden Ausführungen wird gezeigt, wie eine Vektorisierung der Pivot-Berechnung erreicht werden kann. Die Pivot-Strategie, die vektorisiert wird, ist der Median der Mediane von einem ausgedünnten Array. In der vorgestellten Implementierung werden zuerst 72 zufällige Elemente aus dem zu partitionierenden Array entnommen, die das ausgedünnte Array bilden. Für jede Neunergruppe im ausgedünnten Array wird der Median berechnet. Aus den resultierenden acht Medianen

[15] Zur Beschleunigung der Ausführungszeit wird auch in anderen vektorisierten Sortieralgorithmen die Technik der Simulation von breiteren Vektorregistern eingesetzt. Im vektorisierten Bitonic Mergesort aus Quelltext A.8 auf Seite 83 wird eine Registerbreite von 16 Elementen simuliert. Beim vektorisierten Mergesort in Quelltext A.9 auf Seite 84 wird dagegen eine Registerbreite von 32 Elementen simuliert.

wird der Durchschnitt zwischen dem viertgrößten und dem fünftgrößten Element als Pivot-Wert bestimmt.[16]

Die Elemente werden mit Gather-Instruktionen geladen. Die zufälligen Indices für die Gather-Instruktionen erzeugt ein vektorisierter Zufallszahlengenerator. Zur Berechnung der Neuner-Mediane dient die erste Phase der modulbasierten Vektorisierungsstrategie für Sortiernetzwerke aus Abschnitt 4.4.2 auf Seite 35, welche die Elemente spaltenweise sortiert. Statt alle Spalten zu sortieren, werden jedoch die Mediane mit einem Median-Netzwerk ermittelt. Die acht Mediane befinden sich nach der Anwendung des Median-Netzwerks im fünften Vektor des ausgedünnten Arrays. Der Vektor mit den Medianen wird sortiert und der Durchschnitt zwischen dem viertgrößten und dem fünftgrößten Element als Pivot-Wert festgelegt.

6.4.2 Auswahl zufälliger Elemente des Arrays

Zufallszahlengenerator Xoroshiro128+

Der Zufallszahlengenerator Xoroshiro128+[17] ist der Nachfolger von Xorshift128+, einem Zufallszahlengenerator, der in den JavaScript Interpretern von Webbrowsern wie Chrome, Firefox, Safari und Edge verwendet wird. Xoroshiro128+ hat eine Periodenlänge von $2^{128}-1$ und ist einer der schnellsten Zufallszahlengeneratoren (vgl. Kneusel 2018, S. 53–55). Eine mögliche Implementierung von Xoroshiro128+ zeigt Quelltext 6.4.

Quelltext 6.4: Skalarer Xoroshiro128+

```
#include <iostream>

inline uint64_t rotl(const uint64_t x, int k) {
  return (x << k) | (x >> (64 - k)); }                    /* rotieren */

inline uint64_t next(uint64_t& s0, uint64_t& s1) {
  s1 ^= s0;                                    /* s1 und s0 modifizieren */
  s0 = rotl(s0, 24) ^ s1 ^ (s1 << 16);
  s1 = rotl(s1, 37);
  return  s0 + s1; }                  /* nächste Zufallszahl zurückgeben*/

int main() {
  uint64_t s0 = 1314472907419283471;                          /* Seed 1 */
  uint64_t s1 = 7870872464127966567;                          /* Seed 2 */
  for (int i = 0; i < 10; ++i) {               /* 10 zufällige uint64_t */
    std::cout << next(s0, s1) << std::endl; }}
```

[16] Die verwendete Pivot-Strategie dient der Veranschaulichung der Art und Weise wie die Vektorisierung erfolgen kann. Eine optimierte Pivot-Strategie könnte abhängig von der Größe des zu partitionierenden Arrays dynamisch die Größe des ausgedünnten Arrays anpassen.

[17] Der Name Xoroshiro resultiert aus den verwendeten Operationen: XOR, rotate, shift, rotate.

Die Funktion `next` in Zeile 6 generiert 64 zufällige Bits. Die beiden Zustände (*states*) `s1` und `s2` werden per Referenz übergeben und modifiziert, sodass beim nächsten Aufruf von `next` sich eine andere Zufallszahl ergibt. Der Xoroshiro128+ aus Quelltext 6.4 auf Seite 59 verwendet im Unterschied zum Original von Blackman und Vigna (2018) keine globalen Variablen, sodass der Zufallszahlengenerator auch für Multithreading geeignet ist. Die *Seeds* in Zeilen 13 und 14 für die Zustände können als Zufallszahlen, z. B. mit dem C++ Zufallszahlengenerator erzeugt werden. Bei Multithreading kann ergänzend die Thread-ID zu den Anfangs-*Seeds* hinzuaddiert werden, um unterschiedliche *Seeds* zu erreichen. Es ist sicherzustellen, dass kein Anfangs-*Seed* 0 ist. Ferner besteht die Möglichkeit mit speziellen Sprungfunktionen (*jump functions*) Anfangszustände zu erzeugen, die 2^{64} bzw. 2^{96} Aufrufe von `next` auseinander liegen (siehe Blackman und Vigna 2018). Durch die auseinander liegenden Anfangszustände kann garantiert werden, dass die *Streams* von Zufallszahlen in verschiedenen Threads keine gemeinsamen Teil-*Streams* besitzen.

Vektorisierung des Zufallszahlengenerators Xoroshiro128+

Der vektorisierte Zufallszahlengenerator Xoroshiro128+ in Quelltext 6.5 auf der nächsten Seite führt die gleichen Operationen auf den einzelnen Elementen des Vektors aus wie die skalare Version aus Quelltext 6.4 auf Seite 59. Statt 64 zufällige Bits wie in der skalaren Implementierung erzeugt der vektorisierte Zufallszahlengenerator 256 zufällige Bits. Die zwei *Seeds* in den Zeilen 21 und 23, die an `vnext` per Referenz übergeben werden, sind Vektoren mit jeweils vier Zufallszahlen.

Ziel des Zufallszahlengenerators bei der Berechnung des Medians der Mediane ist die Generierung von zufälligen Indices des zu partitionierenden Arrays. Die Indices liegen in einem bestimmten Bereich. Die Funktion `vnext` gibt jedoch zufällige Bits und keine zufälligen Zahlen in einem vorgegebenen Bereich aus. Mithilfe der Funktion `rnd_epu32` in Zeile 15 können die zufälligen 256 Bits in acht 32-Bit-Zufallszahlen in einem Bereich zwischen 0 und einer beliebigen positiven Zahl transformiert werden.[18] Um beispielsweise Zufallszahlen zwischen 100 und 999 zu generieren, werden Zufallszahlen von 0 bis 899 generiert und 100 hinzuaddiert (vgl. Zeilen 25 bis 31 in Quelltext 6.5 auf der nächsten Seite).

Die Initialisierung der *Seeds* könnte mit einer Sprungfunktion erfolgen, falls garantiert werden soll, dass die vier *Streams* von Zufallszahlen im vektorisierten Xoroshiro128+ keine gemeinsamen Teil-*Streams* besitzen. Bei einer zufälligen Initialisierung der *Seeds* ohne Sprungfunktion beträgt die Wahrscheinlichkeit, dass *s Streams*

[18] Die Funktion `rnd_epu32` zur Einschränkung des Bereichs der Zufallszahlen stammt von Lemire (2018).

der Länge l und einer Periode des Zufallszahlengenerators r keine gemeinsamen Teil-*Streams* besitzen, in etwa $p = (1 - \frac{sl}{r})^{s-1}$ (vgl. Kneusel 2018, S. 174). Bei den vektorisierten Xoroshiro128+ ist $s = 4$ und $r = 2^{128}-1$. Wenn beispielsweise $4 \cdot 10^{12}$ 64-Bit-Zufallszahlen generiert werden, dann ist $l = 10^{12}$. Die Wahrscheinlichkeit, dass keine Überlappungen der *Streams* vorliegen, ist bei zufälliger Initialisierung der *Seed*-Vektoren $(1 - \frac{4 \cdot 10^{12}}{2^{128}-1})^{4-1} \approx 1$. Im Beispiel mit $4 \cdot 10^{12}$ Zufallszahlen ist somit die Wahrscheinlichkeit, dass sich *Streams* überlappen, in etwa 0.

Quelltext 6.5: Vektorisierter Xoroshiro128+

```
#include <immintrin.h>
#include <iostream>

#define VROTL(x, k) /* jedes uint64_t im Vektor rotieren */                \
  _mm256_or_si256(_mm256_slli_epi64((x),(k)),_mm256_srli_epi64((x),64-(k)))

inline __m256i vnext(__m256i &s0, __m256i &s1) {
  s1 = _mm256_xor_si256(s0, s1);      /* Vektoren s1 und s0 modifizieren */
  s0 = _mm256_xor_si256(_mm256_xor_si256(VROTL(s0, 24), s1),
                        _mm256_slli_epi64(s1, 16));
  s1 = VROTL(s1, 37);
  return _mm256_add_epi64(s0, s1); }        /* Zufallsvektor zurückgeben */

/* ZZ auf den Bereich zwischen 0 und bound - 1 transformieren */
inline __m256i rnd_epu32(__m256i rnd_vec, __m256i bound) {
  __m256i even = _mm256_srli_epi64(_mm256_mul_epu32(rnd_vec, bound), 32);
  __m256i odd = _mm256_mul_epu32(_mm256_srli_epi64(rnd_vec, 32), bound);
  return _mm256_blend_epi32(odd, even, 0b01010101); }

int main() {
  auto s0 = _mm256_setr_epi64x(8265987198341093849, 3762817312854612374,
                               1324281658759788278, 6214952190349879213);
  auto s1 = _mm256_setr_epi64x(2874178529384792648, 1257248936691237653,
                               7874578921548791257, 1998265912745817298);
  int a = 100, b = 1000;                                  /* 100 <= ZZ < 1000 */
  auto bound = _mm256_set1_epi32(b - a);
  auto a_vec = _mm256_set1_epi32(a)
  for (int i = 0; i < 9; ++i) {          /* 72 ZZ generieren und ausgeben */
    auto result = vnext(s0, s1);      /* Vektor mit 4 zufälligen uint64_t */
    result = rnd_epu32(result, bound);     /* ZZ zwischen 0 und bound - 1 */
    result = _mm256_add_epi32(result, a_vec); /* ZZ zwischen a und b - 1 */
    int* rnds = reinterpret_cast<int *>(&result);/* auf Konsole ausgeben */
    for (int j = 0; j < 8; ++j) { std::cout << rnds[j] << std::endl; }}}
```

6.4.3 Berechnung des Medians der Mediane

Nach dem Laden der 72 zufälligen Werte aus dem zu partitionierenden Array ins ausgedünnte Array wird der Median der Mediane im ausgedünnten Array ermittelt. Hierfür wird die erste Phase der modulbasierten Vektorisierungsstrategie aus

Abschnitt 4.4.2 auf Seite 35 mit einem Median-Netzwerk für 9 Elemente auf das ausgedünnte Array angewandt. Das Median-Netzwerk für 9 Elemente ist in Abb. 6.2 dargestellt.[19]

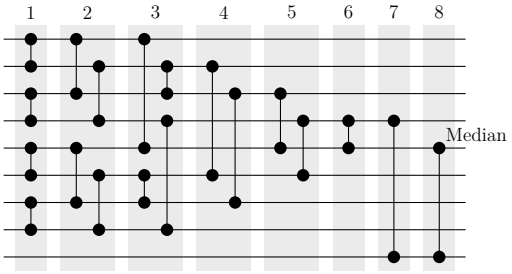


Abb. 6.2: Netzwerk für Median aus 9 Elementen (8 Schritte, 19 Module)

Der Quelltext 6.6 auf der nächsten Seite zeigt die komplette vektorisierte Pivot-Berechnung des Medians der Mediane. Das vektorisierte Median-Netzwerk ist zwischen den Zeilen 21 und 31 implementiert. Damit in jeder Pivot-Berechnung sich unterschiedliche *Seeds* für den Zufallszahlengenerator ergeben, werden die *Seeds* in den Zeilen 10 und 11 in Abhängigkeit von `left` und `right` modifiziert. Die Indices `left` und `right` geben die Grenzen des zu partitionierenden Arrays an. Diese *Seed*-Strategie ist deterministisch, da identische Arrays während allen Partitionierungen die gleichen *Seeds* erhalten.[20] Falls mehr Wert auf die Qualität der *Seeds* gelegt werden soll, dann können diese mit Zufallszahlen initialisiert bzw. verändert werden. Auch Speicheradressen von temporären Variablen können zur Modifizierung und Randomisierung der *Seeds* dienlich sein.

Es kann vorkommen, dass in den 72 zufälligen Werten aus dem zu partitionierenden Array die gleichen Werte vom selben Index geladen werden und somit sich im ausgedünnten Array wiederholen. Außerdem besteht eine geringe Wahrscheinlichkeit, dass sich Teil-*Streams* überlappen können, da die *Seeds* ohne Sprungfunktionen erzeugt werden. Die Überprüfung und Beseitigung bzw. Verhinderung dieser Anomalien würde sich insgesamt negativ auf die Ausführungsgeschwindigkeit von Quicksort auswirken. Ferner ist eine entartete Partitionierung in Einzelfällen unkritisch, da beim nichtquadratischen Quicksort eine alternative Pivot-Strategie zur Verfügung steht, mit der eine Laufzeit im Worst-Case von $2 \cdot \mathcal{O}(32 \cdot n)$ nicht überschritten wird.

Nach der Anwendung des vektorisierten Median-Netzwerks befinden sich die acht Mediane im fünften Vektor des ausgedünnten Arrays. In Zeile 33 werden die acht Mediane sortiert. Der Pivot-Wert ist der Durchschnitt aus dem viert- und fünftgrößten Wert der acht Mediane. Der resultierende Pivot-Wert kann eine Zahl sein, die im

[19] Das Median-Netzwerk für 9 Elemente stammt von Li et al. (2014).

[20] Für Debug- und Optimierungszwecke des Algorithmus ist eine deterministische *Seed*-Strategie vorteilhaft.

Array nicht vorhanden ist. Im Gegensatz zum gewöhnlichen Quicksort wird im nichtquadratischen Quicksort nach der Partitionierung der Pivot-Wert an seine feste Position im Array nicht gesetzt, sodass der Pivot-Wert eine beliebige Zahl sein kann.

Quelltext 6.6: Vektorisierte Pivot-Berechnung

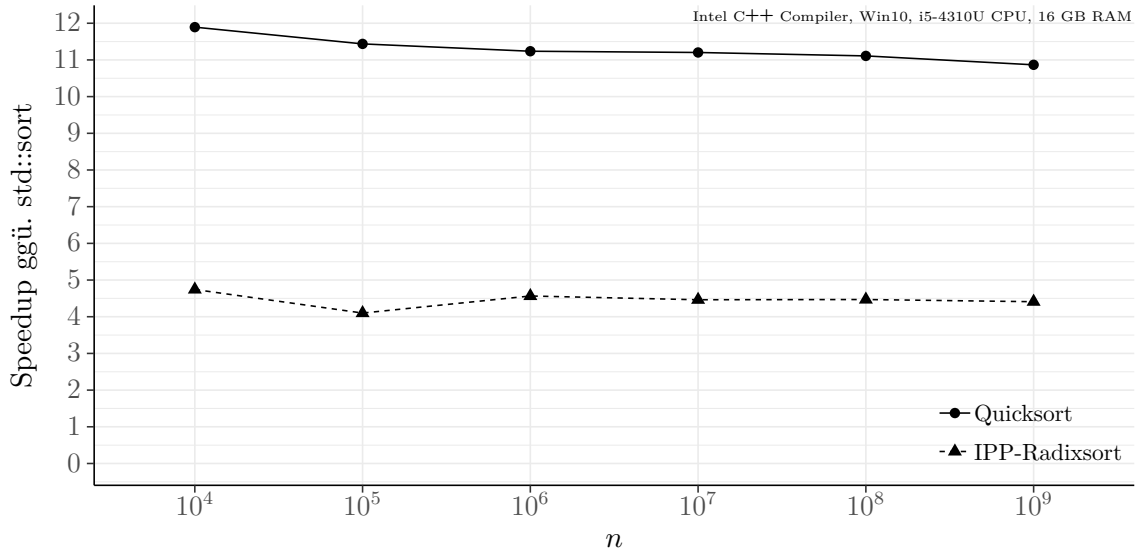
```
inline int get_pivot(int *arr, const int left, const int right) {
  auto bound = _mm256_set1_epi32(right - left + 1);
  auto left_vec = _mm256_set1_epi32(left);

  /* Seeds für vektorisierten Zufallszahlengenerator */
  auto s0 = _mm256_setr_epi64x(8265987198341093849, 3762817312854612374,
                               1324281658759788278, 6214952190349879213);
  auto s1 = _mm256_setr_epi64x(2874178529384792648, 1257248936691237653,
                               7874578921548791257, 1998265912745817298);
  s0 = _mm256_add_epi64(s0, _mm256_set1_epi64x(left));      /* s0 + left */
  s1 = _mm256_sub_epi64(s1, _mm256_set1_epi64x(right));    /* s1 - right */

  __m256i v[9];             /* ausgedünntes Array für Median der Mediane */
  for (int i = 0; i < 9; ++i) {           /* ausgedünntes Array befüllen */
    auto result = vnext(s0, s1);     /* Vektor mit 4 zufälligen uint64_t */
    result = rnd_epu32(result, bound);    /* ZZ zwischen 0 und bound - 1 */
    result = _mm256_add_epi32(result, left_vec);    /* 8 Indices für arr */
    v[i] = _mm256_i32gather_epi32(arr, result, sizeof(uint32_t)); }

  /* Median-Netzwerk für 9 Elemente */
  COEX_VERTICAL(v[0], v[1]); COEX_VERTICAL(v[2], v[3]);     /* Schritt 1 */
  COEX_VERTICAL(v[4], v[5]); COEX_VERTICAL(v[6], v[7]);
  COEX_VERTICAL(v[0], v[2]); COEX_VERTICAL(v[1], v[3]);     /* Schritt 2 */
  COEX_VERTICAL(v[4], v[6]); COEX_VERTICAL(v[5], v[7]);
  COEX_VERTICAL(v[0], v[4]); COEX_VERTICAL(v[1], v[2]);     /* Schritt 3 */
  COEX_VERTICAL(v[5], v[6]); COEX_VERTICAL(v[3], v[7]);
  COEX_VERTICAL(v[1], v[5]); COEX_VERTICAL(v[2], v[6]);     /* Schritt 4 */
  COEX_VERTICAL(v[3], v[5]); COEX_VERTICAL(v[2], v[4]);     /* Schritt 5 */
  COEX_VERTICAL(v[3], v[4]);                                /* Schritt 6 */
  COEX_VERTICAL(v[3], v[8]);                                /* Schritt 7 */
  COEX_VERTICAL(v[4], v[8]);                                /* Schritt 8 */

  SORT_8(v[4]);                    /* die 8 Mediane in v[4] sortieren */
  return average(_mm256_extract_epi32(v[4], 3),            /* Pivot-Wert */
                 _mm256_extract_epi32(v[4], 4)); }
```

6.5 Performanz des vektorisierten Quicksort

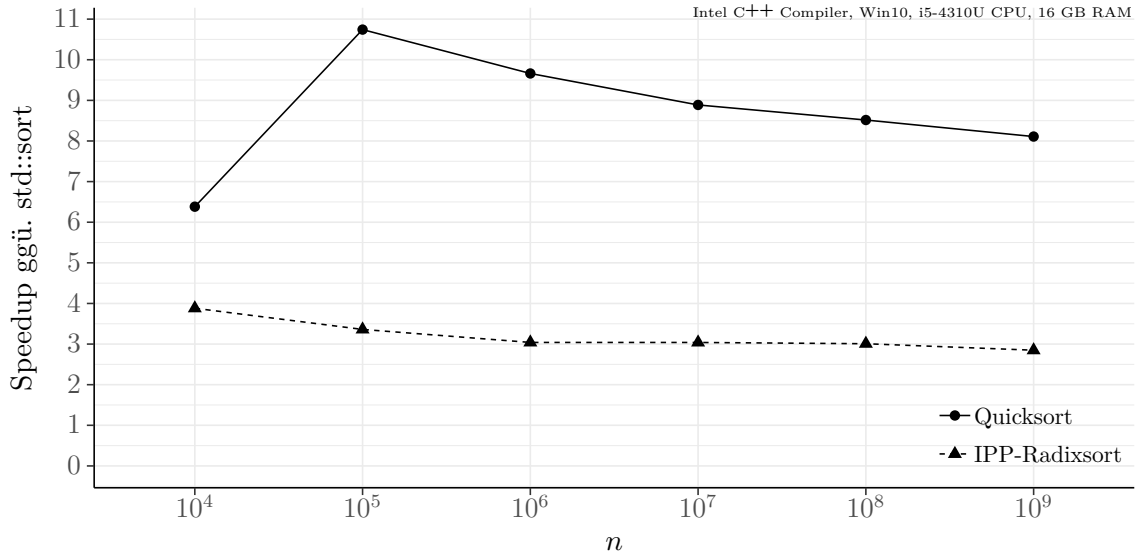
Der zur Messung der Performanz verwendete Sortieralgorithmus ist ein Hybrid aus vektorisierten Sortiernetzwerken[21] und vektorisiertem nichtquadratischen Quicksort. Die Pivot-Strategie im nichtquadratischen Quicksort wird ergänzt um die vektorisierte Berechnung des Medians der Mediane. Die rekursive Funktion dieses Sortieralgorithmus ist im Quelltext A.14 auf Seite 89 enthalten. Vereinfachend wird im Folgenden

[21] Teilarrays, die weniger als 513 Elemente enthalten, werden mit Sortiernetzwerken sortiert.

der entworfene und implementierte Sortieralgorithmus als (vektorisierter) Quicksort bezeichnet. In Abb. 6.3 ist der Speedup des vektorisierten Quicksort ggü. `std::sort` dargestellt. Zum Vergleich ist ebenfalls der Beschleunigungsfaktor von IPP-Radixsort ggü. `std::sort` abgetragen.



(a) Zufallszahlen zwischen `INT32_MIN` und `INT32_MAX`



(b) Zufallszahlen zwischen 1 000 000 und 1 000 100

Abb. 6.3: Speedup des vektorisierten Quicksort ggü. std::sort

Für gleichverteilte Zufallszahlen im kompletten Interval des Datentyp `int` ist der Speedup ggü. `std::sort` in Abhängigkeit von der Arraylänge zwischen 11 und 12. Der Speedup des vektorisierten Quicksort ggü. `std::sort` ist höher als die Anzahl an Elementen im Vektorregister.[22] Dies liegt daran, dass der Algorithmus eine

[22]Der Speedup hängt vom Compiler und der jeweiligen Implementierung von `std::sort` (Introsort)

Registerbreite von 64 Elementen simuliert, was sowohl die Latenzen einzelner Instruktionen vermindert als auch den Speicherzugriff effizienter macht. Im Vergleich zu IPP-Radixsort sortiert der vektorisierte Quicksort Arrays mit gleichverteilten 32-Bit-Integern mehr als doppelt so schnell.

Um den Speedup für Arrays mit einer hohen Anzahl an doppelten Werten zu ermitteln, werden die Arrays mit Zufallszahlen aus einem engen Bereich befüllt. In Abb. 6.3b auf Seite 64 sind exemplarisch die Ergebnisse für Arrays, die mit Zufallszahlen aus dem Bereich zwischen 1 000 000 und 1 000 100 befüllt sind, dargestellt. Für 10^4 Elemente ist der Beschleunigungsfaktor ggü. `std::sort` in etwa 6,5. Da der vektorisierte Quicksort für weniger als 513 Elemente Sortiernetzwerke verwendet, müssen mehr als 512 Elemente im Array identisch sein, damit die Rekursion für diese vorzeitig abbrechen kann. Bei 10^4 Zufallszahlen aus dem Bereich zwischen 1 000 000 und 1 000 100 sind wahrscheinlich keine Zahlen vertreten, die mehr als 512 mal vorkommen, deshalb werden diese zum Schluss der Rekursion mit Sortiernetzwerken sortiert, was einen relativ niedrigen Beschleunigungsfaktor ergibt. Bei der Sortierung von Arrays mit 10^5 bis 10^9 zufällig verteilten Elementen aus einem engen Intervall fällt der Beschleunigungsfaktor von etwa 10,7 auf 8.

Die geringste Beschleunigung erzielt der vektorisierte Quicksort für bereits aufsteigend sortierte Arrays (im Diagramm nicht dargestellt). Bei bereits aufsteigend sortierten Arrays ist der Speedup ca. 1,4 ggü. `std::sort`.[23] Der vektorisierte Quicksort enthält keine Optimierungen bezüglich bereits sortierter Arrays bzw. Teilarrays, deshalb ist die Rechenzeit unverändert zu unsortierten Arrays. Der von Intel implementierte `std::sort` ist dagegen bei bereits sortierten Arrays effizienter als bei unsortierten Arrays.

6.6 Exkurs: Vektorisierung von Quickselect

Quickselect ist ein Selektionsalgorithmus. In Quickselect wird das Array entsprechend einer Position k umgeordnet, sodass an der k-ten Position das Element steht, welches sich dort ebenfalls beim komplett sortierten Array befinden würde. Ergänzend gilt nach Ausführung des Algorithmus, dass alle Elemente links von dem k-ten Element kleiner-gleich dem k-ten Element und alle Elemente rechts von dem k-ten Element größer-gleich dem k-ten Element sind. Unter anderem kann Quickselect für die

ab. In allen vorgestellten Messungen wird der Intel-Compiler in Windows verwendet. Beim Compiler von GCC beträgt der Speedup in etwa 9,5 für die Konfiguration in Abb. 6.3a auf Seite 64. Für die Konfiguration in Abb. 6.3b auf Seite 64 ist dagegen bei GCC der Beschleunigungsfaktor des vektorisierten Quicksort höher als in der Abbildung.

[23] Der Speedup von Intels IPP-Radixsort beträgt für bereits sortierte Arrays in etwa 0,5 ggü. `std::sort`, d. h. bei sortierten Arrays ist IPP-Radixsort langsamer als `std::sort`.

Berechnung von Quantilen wie den Median benutzt werden. In Quickselect wird die gleiche Partitionierungsfunktion wie beim Quicksort verwendet. Im Durchschnitt hat Quickselect eine Komplexität von $\mathcal{O}(n)$. Im Worst-Case ist die Komplexität von Quickselect jedoch wie bei Quicksort $\mathcal{O}(n^2)$ (siehe Heun 2003, S. 87 ff.).

Mit der gleichen Pivot-Strategie wie beim nichtquadratischen Quicksort und derselben Vorgehensweise im Umgang mit einer hohen Anzahl an wiederholenden Werten im Array kann ein nichtquadratischer Quickselect implementiert werden. Der nichtquadratische Quickselect hat wie der nichtquadratische Quicksort im Worst-Case eine Laufzeit von $2 \cdot \mathcal{O}(32 \cdot n)$. Zur Vektorisierung von Quickselect ist lediglich die Anpassung der rekursiven Funktion des vektorisierten Quicksort erforderlich. Die vektorisierte Partitionierung und die vektorisierte Pivot-Berechnung sind beim vektorisierten Quickselect mit der von vektorisiertem Quicksort identisch.[24]

Die Funktion `std::nth_element` in der C++ STL ordnet die Elemente im Array[25] wie bei Quickselect beschrieben um (vgl. C++ Reference 2017). In Abb. 6.4 ist der Speedup des vektorisierten Quickselect ggü. `std::nth_element` zur Berechnung des Medians aus mit Zufallszahlen befüllten Arrays dargestellt. Der Speedup des vektorisierten Quickselect ggü. `std::nth_element` liegt zwischen 7 und 12.[26]

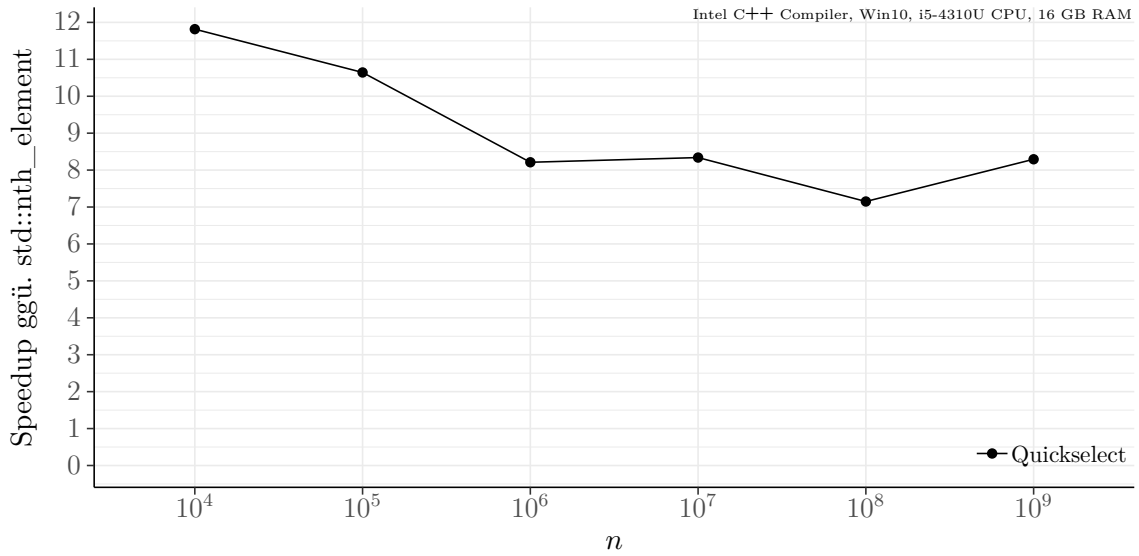


Abb. 6.4: Speedup des vektorisierten Quickselect ggü. std::nth_element

[24] Die Rekursion für den vektorisierten Quickselect ist im Quelltext A.15 auf Seite 90 enthalten.

[25] Der Begriff Array wird zur Vereinfachung verwendet. Die Parameter der Funktion `std::nth_element` sind generische *Random-access* Iteratoren.

[26] Auch andere Sortier-Strategien wie `std::partial_sort` können vektorisiert werden. Hierfür zuerst mit vektorisiertem Quickselect bis zum k-ten Element das Array aufteilen und anschließend bis zum k-ten Element das Array mit vektorisiertem Quicksort sortieren.

7 Weiterführende Diskussion

7.1 Sortierung beliebiger einfacher Datentypen

Die vorgestellten Algorithmen sortieren Arrays von Datentyp `int`. C++ enthält jedoch vorzeichenbehaftete und vorzeichenlose Integer für 8, 16, 32 und 64 Bit bzw. Gleitkommazahlen für 32 und 64 Bit. Die Vektorisierung von Sortiernetzwerken, Mergesort und Quicksort ist für alle einfachen Datentypen in AVX2 möglich, da die benötigten intrinsischen Funktionen hierfür vorhanden sind.[1]

Die Vektorisierung von Quicksort benötigt in AVX2 eine Lookup-Tabelle mit Permutations-Masken zur Partitionierung einzelner Vektoren. Die Größe der Lookup-Tabelle wächst exponentiell mit der Anzahl an Elementen im Vektor. Bei 32 Elementen im Vektor, enthielte die Tabelle $2^{32} = 4\,294\,967\,296$ Einträge. Für mehr als 16 Elemente im Vektor wäre die Vektorisierung von Quicksort mit einer Lookup-Tabelle ineffizient. Die Grenze von 16 Elementen im Vektor wird bei AVX2 für die Partitionierung nicht überschritten, da beispielsweise für den Datentyp `int8_t` lediglich 128-Bit-Vektoren zur Partitionierung eines einzelnen Vektors verwendet werden können, da keine Permutations-Befehle für 256-Bit-Vektoren für diesen Datentyp existieren.[2]

Statt für jeden einfachen Datentyp separate vektorisierte Sortierfunktionen zu erstellen, können Funktionen für Datentypen, die über dieselbe Anzahl an Bits verfügen, wiederverwendet werden. Beispielsweise kann ein Sortieralgorithmus für den Datetyp `int` auch Arrays mit `unsigned int` bzw. `float` sortieren. Sind alle `unsigned int` im Array kleiner-gleich `INT32_MAX` bzw. sind alle `float` im Array größer-gleich 0, dann genügt es den Array-Pointer auf `int*` vor der Sortierung zu

[1] Teilweise sind jedoch Konvertierungen und/oder `reinterpret_cast` notwendig, um die Instruktionen von anderen Datentypen zu verwenden. Beispielsweise gibt es in AVX2 keine intrinsische Funktion, die das paarweise Minimum oder Maximum zwischen den Elementen von zwei Vektoren für 64-Bit-Integer berechnet. Erst bei AVX-512 existieren solche Funktionen (`_mm256_min_epi64` bzw. `_mm256_max_epi64`). Auch ist nicht immer in AVX2 die Verwendung von 256-Bit-Vektoren möglich. Für 8 und 16 Bit Integer müssten in einigen Fällen 128-Bit-Vektoren verwendet werden, da nicht alle benötigten Instruktionen für 256-Bit-Vektoren vorhanden sind (z. B. fehlen Permutations-Befehle).

[2] Die Lookup-Tabelle zur effizienten Vektorisierung von Quicksort ist eine Notwendigkeit bei AVX2, die in künftigen Instruktionssätzen (bzw. in AVX-512) entfällt, da anstelle der Lookup-Tabelle Komprimierungsbefehle verwendet werden können.

reinterpretieren. Für Arrays mit beliebigen Ausprägungen von `unsigned int` bzw. `float` sind in Quelltext 7.1 Konvertierungsfunktionen angegeben. Um ein Array mit `unsigned int` Elementen zu sortieren, genügt es, von jedem Element im Array vor der Sortierung `2147483648u` zu subtrahieren. Nach der Sortierung muss für jedes Element `2147483648u` hinzuaddiert werden, damit die ursprünglichen Zahlen wiederhergestellt sind (vgl. Konvertierungsfunktionen in Zeilen 2 und 6). Bei der Sortierung eines Arrays von Datentyp `float` mit einer Sortiermethode für `int` kann vor und nach der Sortierung dieselbe Transformation angewandt werden (siehe Zeile 10). Falls das Vorzeichenbit beim `float` gesetzt ist, dann werden vor und nach der Sortierung außer dem Vorzeichenbit alle Bits der Zahl invertiert.

Quelltext 7.1: Konvertierungen für Sortierung mit Datentyp int

```
1  /* vor der Sortierung von unsigned int aufrufen */
2  inline void convert_unsigned_before(unsigned int *arr, int n) {
3    for (int i = 0; i < n; ++i) { arr[i] = arr[i] - 2147483648u; }}
4
5  /* nach der Sortierung von unsigned int aufrufen */
6  inline void convert_unsigned_after(unsigned int *arr, int n) {
7    for (int i = 0; i < n; ++i) { arr[i] = arr[i] + 2147483648u; }}
8
9  /* vor und nach der Sortierung von float aufrufen */
10 inline void convert_float(float *arr, int n) {
11   for (int i = 0; i < n; ++i) {
12     int& x = reinterpret_cast<int&>(arr[i]);
13     x = x ^ ((x >> 31) &  0x7FFFFFFF); }}
```

Zur Verminderung der Ausführungszeit kann die Konvertierung der einzelnen Elemente direkt während der ersten Partitionierung von Quicksort vektorisiert erfolgen. Nach Anwendung der Sortiernetzwerke, vor dem Abspeichern der Vektoren können die Datenelemente wieder in den ursprünglichen Zustand konvertiert werden.

7.2 Sortierung von komplexen Datentypen

Komplexe Datentypen für die Sortierung sind im Wesentlichen Strukturen, Unions oder Klassen. Im Folgenden wird der Begriff Objekt verwendet, um eine Instanz eines komplexen Datentyps kenntlich zu machen. Für einfache Datentypen ist die Sortierung mit SIMD-Vektoren unproblematisch, da für die jeweiligen Datentypen passende Instruktionen auf modernen Architekturen zur Verfügung stehen. Falls Objekte die Größe von einfachen Datentypen nicht überschreiten, kann die vektorisierte Sortierung ohne Anpassung des Algorithmus erfolgen. Hierfür muss der numerische Sortierschlüssel in den höheren Bits des Objekts repräsentiert sein. Möglich ist auch, ganz auf komplexe Datentypen während der Sortierung zu verzichten, falls

die zu sortierende Informationseinheit bitweise als numerischer Wert eines einfachen Datentyps codiert werden kann.

Eine Methode zur Sortierung von größeren Objekten, ohne eine Anpassung des vektorisierten Algorithmus vorzunehmen, besteht darin, den Sortierschlüssel mit dem zugehörigen Index des Objekts im Array in einer Zahl zu vereinigen und in ein separates Array zu speichern, das separate Array mit den Sortierschlüssel-Index-Paaren zu sortieren und anschließend die Objekte entsprechend den Indices umzuordnen. Wird ein Array mit Sortierschlüssel-Index-Paaren sortiert, dann ist die Sortierung stabil auch bei einem instabilen Sortierverfahren, da bei identischen Sortierschlüsseln der Index ausschlaggebend ist, in welcher Reihenfolge die Elemente nach der Sortierung vorliegen. Durch den verstreuten Zugriff auf den Speicher während des Umsortierens der Objekte generiert diese Methode vermehrt Cache-Fehler. Darüber hinaus ist das Prefetching des Prozessors beim randomisierten Speicherzugriff ineffizient (vgl. Inoue und Taura 2015, S. 1275).

Für einzelne vektorisierte Sortieralgorithmen können auch spezifische Lösungen zur Sortierung von Objekten implementiert werden. Beispielsweise entwickeln Inoue und Taura (2015) für den vektorisierten Multiway Mergesort ein Verfahren, welches statt Sortierschlüssel-Index-Paare Sortierschlüssel-*StreamID*-Paare erzeugt. Da Inoue und Taura 32 *Streams* verwenden, kann der Sortierschlüssel mehr Bits umfassen. Ferner werden die Daten nach jedem 32-Wege Multiway Merge umgeordnet und nicht erst zum Schluss der Sortierung, was die Cache-Effizienz verbessert.

Gueron und Krasnov (2016) vermeiden die Umsortierung der Objekte beim vektorisierten Quicksort gänzlich, indem sie ein Array aus Zeigern sortieren. Mit Gather-Befehlen werden anhand der Zeiger die Sortierschlüssel aus den Datensätzen entnommen. Für den Vektor mit Sortierschlüsseln wird die Permutations-Maske ermittelt und entsprechend dieser Maske die Zeiger partitioniert. Nach der Sortierung sind nicht die Datensätze, sondern die Zeiger zu den Datensätzen im Speicher geordnet. Wenn im Anschluss sequentiell auf die Objekte zugegriffen wird, dann ist der Speicherzugriff ineffizient, da die Objekte verstreut im Speicher vorliegen.

Anstelle von Objekten sortiert Bramas (2017) Schlüssel/Wert-Paare mit dem vektorisierten Hybridalgorithmus aus Quicksort und Sortiernetzwerken. Die Schlüssel und Werte sind in zwei unterschiedlichen Arrays gespeichert. Als Werte können auch Zeiger auf Objekte verwendet werden. In der Quicksort-Phase werden die für die Schlüssel berechneten Partitionierungs-Masken sowohl auf das Array mit Schlüsseln als auch das Array mit Werten angewandt. In der Sortiernetzwerk-Phase finden die gleichen Permutationen, die auf dem Vektor mit Schlüsseln ausgeführt werden,

ebenfalls auf dem Vektor mit Werten statt. Diese Methode eignet sich insbesondere für Datensätze, die als *Structure of Arrays* organisiert sind.[3]

Die Art und Weise wie Objekte möglichst effizient sortiert werden können, ist somit von der Größe des zugrunde liegenden Objekts und vom verwendeten vektorisierten Sortierverfahren abhängig. Auch können statt den Objekten Datensätze, die als *Structure of Arrays* vorliegen, vektorisiert sortiert werden. Im Optimalfall passt die komplette Informationseinheit in einen einfachen Datentyp, sodass zusätzlicher Speicher, Umsortierungen der Objekte bzw. duplizierte Permutationen entfallen.

7.3 Parallelisierung

Die Techniken zur Parallelisierung von skalaren Sortieralgorithmen können auf vektorisierte Sortieralgorithmen übertragen werden. Ein mögliches Vorgehen zur Parallelisierung von Quicksort mit OpenMP ist in Quelltext 7.2 dargestellt.

Quelltext 7.2: Parallelisierung von Quicksort mit OpenMP

```
1  void qs_core(int *arr, int left, int right) {
2    if (left < right) {
3      int bound = partition(arr, left, right, arr[right]);
4      swap(arr[bound], arr[right]);      /* Pivot ist letzter Wert in arr */
5  #pragma omp task final(bound - left < 2000) mergeable
6      { qs_core(arr, left, bound - 1); }              /* linkes Teilarray */
7      qs_core(arr, bound + 1, right); }}             /* rechtes Teilarray */
8
9  void quicksort(int *arr, int n) {
10 #pragma omp parallel shared(arr, n)                 /* parallele Region */
11 #pragma omp single nowait                  /* ein Thread generiert Tasks */
12   { qs_core(arr, 0, n - 1); }}
```

Beim ersten Aufruf von `qs_core` findet die Partitionierung des kompletten Arrays sequentiell statt. In den nachfolgenden `qs_core` Aufrufen partitionieren mehrere Threads verschiedene Teilarrays. Während der Erstellung eines Tasks für das linke Array in Zeile 5 wird bereits das rechte Teilarray partitioniert. Es ist effizienter, ein Task für das linke Teilarray statt zwei separate Tasks für die beiden Teilarrays auszuführen (siehe Ruud 2017, S. 122 ff.). Für kleine Teilarrays übersteigt der Mehraufwand für die Erstellung und Verwaltung von Tasks die zeitliche Einsparung durch Parallelisierung. In Zeile 5 wird deshalb die Klausel `final` verwendet, um linke Teilarrays mit weniger als 2 000 Elementen sequentiell zu sortieren. Die optimale Schwelle zum Übergang zur sequentiellen Sortierung ist experimentell zu ermitteln.

[3] In *Structure of Arrays* sind die einzelnen Felder eines Datensatzes auf unterschiedliche Arrays verteilt. Bestehen beispielsweise die Datensätze aus den Feldern Alter und Geschlecht, dann sind bei *Structure of Arrays* alle Altersangaben hintereinander im Speicher in einem Array und die Geschlechtsangaben in einem anderen Array gespeichert.

8 Fazit

Moderne Prozessoren verfügen über Vektorinstruktionen, mit denen Arrays aus einfachen Datentypen vektorisiert sortiert werden können. In dieser Arbeit wurden Algorithmen, Vorgehensweisen und Optimierungen zur Vektorisierung von Sortiernetzwerken, Mergesort und Quicksort vorgestellt. Die Implementierung der vektorisierten Verfahren erfolgte in AVX2 mit intrinsischen Funktionen für den Datentyp `int`.

Für die Sortierung von kurzen Arrays eignen sich vektorisierte Sortiernetzwerke. Bei längeren Arrays sinkt die Performanz der vektorisierten Sortiernetzwerke aufgrund der höheren algorithmischen Komplexität im Vergleich zu Mergesort bzw. Quicksort. Den höchsten Speedup gewährleisten Vorgehensweisen zur Vektorisierung, bei denen alle Elemente des SIMD-Vektors miteinander unverbundene Module des Sortiernetzwerks repräsentieren, und somit die Vergleichs- und Austauschoperationen zwischen den Vektoren keine redundanten Module enthalten. Die regelmäßige Struktur von Bitonic Mergesort lässt sich algorithmisch als iterative Berechnungsvorschrift vektorisieren. Falls Arrays als Matrix in zeilendominierter Reihenfolge aufgefasst werden, dann können zur Sortierung der Spalten beliebige Sortiernetzwerke verwendet werden, auch unregelmäßige. Zum Mergen der sortierten Spalten kann ein vektorisierter Bitonic Merge implementiert werden. Eine vorherige Transponierung der Matrix ist dabei nicht erforderlich. Vektorisierte Sortiernetzwerke benötigen keinen zusätzlichen Speicher, jedoch können diese durch die Sortierung in einem zusätzlichen ausgerichteten Speicher beschleunigt werden.

Der vektorisierte Mergesort ist ein Hybrid aus Sortiernetzwerken und den skalaren Mergesort. Kleine Teilarrays werden mit vektorisierten Sortiernetzwerken sortiert. Zur Implementierung der vektorisierten Mergeoperation von zwei beliebig langen sortierten Teilarrays wird mindestens ein Mergenetzwerk für zwei Vektoren benötigt. Im Vergleich zu Bitonic Merge hat die vektorisierte Mergeoperation von Mergesort eine bessere asymptotische Komplexität. Die vektorisierte Mergeoperation braucht jedoch zusätzlichen Speicher der Größe n, wobei n die Gesamtanzahl der zu mergenden Elemente ist. Der Speedup des vektorisierten Mergesort gegenüber `std::sort` nimmt mit steigenden Arraylängen ab und beträgt etwa 5,5 bei der Sortierung von 10^9 Elementen.

Der performanteste vektorisierte Sortieralgorithmus dieser Arbeit ist Quicksort.

Bei der Sortierung von Arrays, die mit Zufallszahlen befüllt sind, erreicht der vektorisierte Quicksort einen Speedup zwischen 11 und 12 gegenüber `std::sort`. Der Speedup ist somit höher als die Anzahl an Elementen im Vektorregister. Im Vergleich zu IPP-Radixsort ist der vektorisierte Quicksort mindestens doppelt so schnell. Der vektorisierte Quicksort benötigt keinen linear wachsenden zusätzlichen Speicher und hat keinen quadratischen Worst-Case. Der quadratische Worst-Case wird durch eine vektorisierte Pivot-Strategie basierend auf Median der Mediane und Binary-Quicksort verhindert. Der vektorisierte Quicksort ist ein Hybridalgorithmus der vektorisierte Netzwerke zur Sortierung von kleinen Teilarrays und in der Pivot-Berechnung des Medians der Mediane verwendet. Aufgrund des Mangels an Komprimierungsbefehlen in AVX2 erfolgt die Partitionierung der Vektoren mit Unterstützung von Permutations-Masken, die in einer Lookup-Tabelle gespeichert sind. Falls alle Elemente eines Teilarrays identisch sind, dann wird die Sortierung für dieses Teilarray beenden. Um festzustellen, ob alle Werte in einem Teilarray identisch sind, werden während der Partitionierung die größten und kleinsten Werte des Teilarrays ermittelt. Mit derselben Partitionierungsfunktion wie beim vektorisierten Quicksort kann auch ein vektorisierter nichtquadratischer Quickselect implementiert werden. Der vektorisierte Quickselect erreicht gegenüber `std::nth_element` einen Speedup von 7 bis 12.

Falls ergänzend zum Datentyp `int` andere einfache Datentypen vektorisiert sortiert werden sollen, dann existieren in AVX2 die nötigen Instruktionen hierfür ebenfalls. Es ist nicht erforderlich für einfache Datentypen, die über dieselbe Anzahl an Bits verfügen (z. B. `int`, `float` oder `unsigned int`), separate Implementierungen des Sortieralgorithmus zu erstellen. Eine vektorisierte Transformation der Werte auf dem zu sortierenden Datentyp ist möglich. Bei Sortierung von Objekten, welche die Größe von einfachen Datentypen nicht überschreiten, kann der vektorisierte Sortieralgorithmus ohne Anpassung verwendet werden. Hierfür muss in den obersten Bits des Objekts der numerische Sortierschlüssel enthalten sein. Die Parallelisierung von vektorisierten Sortieralgorithmen kann mit Techniken für Parallelisierung von skalaren Sortieralgorithmen, z. B. mittels Tasks bei OpenMP realisiert werden.

Bereits heute ist es möglich, vektorisierte Sortieralgorithmen zu implementieren, die schneller als skalare Sortieralgorithmen sind. Auf künftigen Prozessoren mit größeren Vektorregistern und vielfältigeren Instruktionen dürfte der Geschwindigkeitsunterschied zugunsten der vektorisierten Sortieralgorithmen weiter zunehmen. Die Vektorisierung von Sortieralgorithmen könnte somit zur effizienteren Ausnutzung der zur Verfügung stehenden Rechenressourcen beitragen und zahlreiche Anwendungen in der Forschung und Praxis beschleunigen.

A Anhang

A.1 Ergänzende Sortiernetzwerke

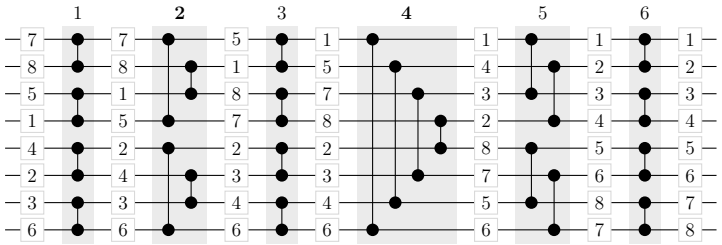


Abb. A.1: Bitonic Mergesort für 8 Elemente (24 Module, 6 Schritte)

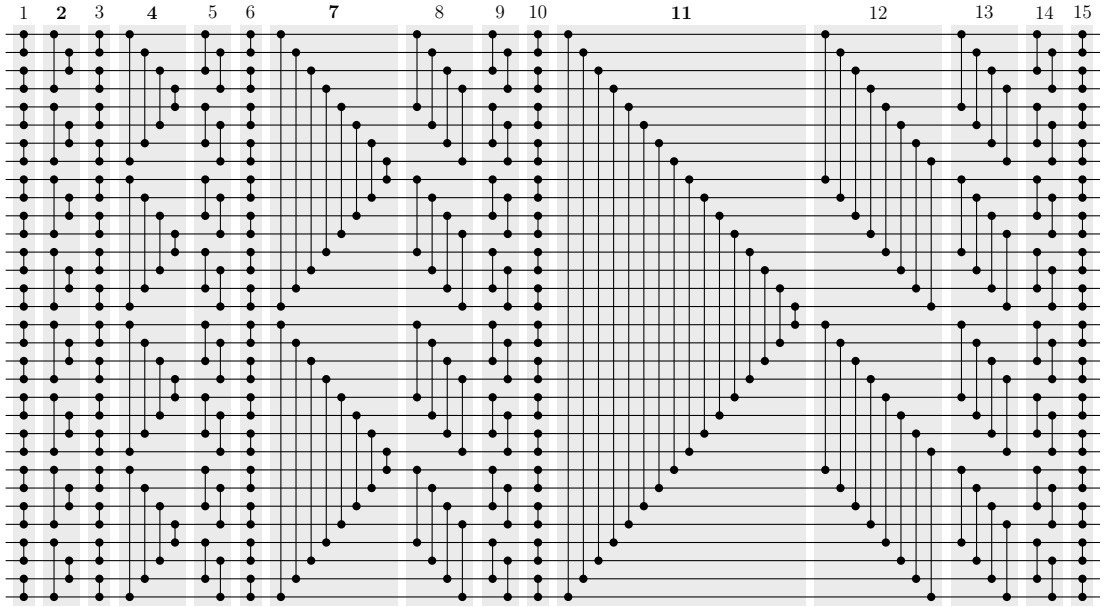


Abb. A.2: Bitonic Mergesort für 32 Elemente (240 Module, 15 Schritte)

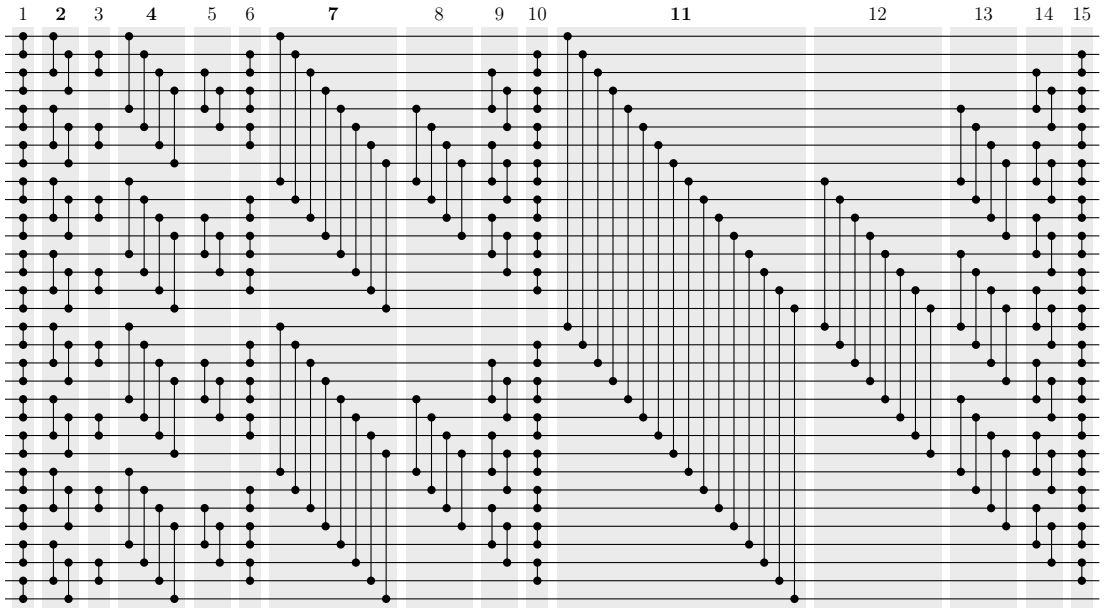


Abb. A.3: Odd-Even Mergesort für 32 Elemente (191 Module, 15 Schritte)

A.2 Quelltext

Quelltext A.1: Sortiernetzwerk für 8 Elemente

```
#include <cstdio>
#include <immintrin.h>

/* lade aus Array 8 int ins Vektorregister */
#define LOAD_VEC(arr) _mm256_loadu_si256(reinterpret_cast<__m256i *>(arr))

/* speichere Vektor ins Array */
#define STORE_VEC(arr, vec)                                                \
  _mm256_storeu_si256(reinterpret_cast<__m256i *>(arr), vec)

/* Blend-Maske aus Permutations-Indices für aufsteigendes Sortieren */
#define ASC(a, b, c, d, e, f, g, h)                                        \
  (((h < 7) << 7) | ((g < 6) << 6) | ((f < 5) << 5) | ((e < 4) << 4) |     \
      ((d < 3) << 3) | ((c < 2) << 2) | ((b < 1) << 1) | (a < 0))

/* Blend-Maske aus Permutations-Indices für absteigendes Sortieren */
#define DESC(a, b, c, d, e, f, g, h)                                       \
  (ASC(a, b, c, d, e, f, g, h) ^ 0b11111111)

/* vektorisiertes compare-exchange mit Permutation */
#define COEX_PERMUTE(vec, a, b, c, d, e, f, g, h, MASK){                   \
    __m256i permute_mask = _mm256_setr_epi32(a, b, c, d, e, f, g, h);      \
    __m256i permuted = _mm256_permutevar8x32_epi32(vec, permute_mask);     \
    __m256i min = _mm256_min_epi32(permuted, vec);                         \
    __m256i max = _mm256_max_epi32(permuted, vec);                         \
    vec = _mm256_blend_epi32(min, max, MASK(a, b, c, d, e, f, g, h));}

/* vektorisiertes compare-exchange mit Shuffle */
#define COEX_SHUFFLE(vec, a, b, c, d, e, f, g, h, MASK){                   \
    constexpr auto shuffle_mask = _MM_SHUFFLE(d, c, b, a);                 \
    __m256i shuffled = _mm256_shuffle_epi32(vec, shuffle_mask);            \
    __m256i min = _mm256_min_epi32(shuffled, vec);                         \
    __m256i max = _mm256_max_epi32(shuffled, vec);                         \
    vec = _mm256_blend_epi32(min, max, MASK(a, b, c, d, e, f, g, h));}

/* Sortiernetzwerk für 8 int mit compare-exchange Makros */
#define SORT_8(vec, ASC_OR_DESC){                                          \
    COEX_SHUFFLE(vec, 1, 0, 3, 2, 5, 4, 7, 6, ASC_OR_DESC);                \
    COEX_SHUFFLE(vec, 2, 3, 0, 1, 6, 7, 4, 5, ASC_OR_DESC);                \
    COEX_SHUFFLE(vec, 0, 2, 1, 3, 4, 6, 5, 7, ASC_OR_DESC);                \
    COEX_PERMUTE(vec, 7, 6, 5, 4, 3, 2, 1, 0, ASC_OR_DESC);                \
    COEX_SHUFFLE(vec, 2, 3, 0, 1, 6, 7, 4, 5, ASC_OR_DESC);                \
```

```
43      COEX_SHUFFLE(vec, 1, 0, 3, 2, 5, 4, 7, 6, ASC_OR_DESC);}
44
45  int main(){
46    int arr[8] = {7, 8, 5, 1, 4, 3, 2, 6};
47    auto vec = LOAD_VEC(arr);
48    SORT_8(vec, ASC); /* sortiere in aufsteigender Reihenfolge */
49    STORE_VEC(arr, vec);
50    for(int i: arr) printf("%d ", i); /* 1 2 3 4 5 6 7 8 */
51    puts("");
52    SORT_8(vec, DESC); /* sortiere in absteigender Reihenfolge */
53    STORE_VEC(arr, vec);
54    for(int i: arr) printf("%d ", i); /* 8 7 6 5 4 3 2 1 */
55    return 0;}
```

Quelltext A.2: Sortiernetzwerk für 16 Elemente

```
1  #define COEX_VERTICAL(a, b){ /* berechne acht Vergleichsmodule */          \
2    __m256i c = a; a = _mm256_min_epi32(a, b); b = _mm256_max_epi32(c, b);}
3
4  #define LAST_3_STEPS(v){ /* letzte drei Schritte des Netzwerks */          \
5    COEX_PERMUTE(v, 4, 5, 6, 7, 0, 1, 2, 3, ASC);                            \
6    COEX_SHUFFLE(v, 2, 3, 0, 1, 6, 7, 4, 5, ASC);  /* Shuffle */            \
7    COEX_SHUFFLE(v, 1, 0, 3, 2, 5, 4, 7, 6, ASC);} /* Shuffle */
8
9  /* merge zwei Vektoren: einer aufsteigend, der andere absteigend sortiert*/
10 #define MERGE_16_ASC(v1, v2){                                              \
11   COEX_VERTICAL(v1, v2); /* Schritt 7, keine Permutation */                \
12   LAST_3_STEPS(v1); LAST_3_STEPS(v2);} /* Schritte 8, 9 und 10 */
13
14 #define SORT_16(v1, v2){ /* sortiere 16 int */                             \
15   SORT_8(v1, ASC); SORT_8(v2, DESC); /* sortiere Vektoren v1 und v2 */     \
16   MERGE_16_ASC(v1, v2);} /* merge v1 und v2 */
```

Quelltext A.3: Bitonic Mergesort für 8 Elemente optimiert

```
1  #include <cstdio>
2  #include <immintrin.h>
3
4  #define COEX_VERTICAL_128i(a, b){ /* berechne vier Vergleichsmodule */    \
5    __m128i c = a; a = _mm_min_epi32(a, b); b = _mm_max_epi32(c, b);}
6
7  /* shuffle 2 Vektoren, Instruktion für int fehlt, deshalb mit float */
8  #define SHUFFLE_TWO_VECS(a, b, mask)                                      \
9    reinterpret_cast<__m128i>(_mm_shuffle_ps(                               \
10       reinterpret_cast<__m128>(a), reinterpret_cast<__m128>(b), mask));
11
```

```
12 int main(){
13   int arr[8] = {11, 29, 13, 23, 37, 17, 19, 31};    /* lade Vektoren */
14   __m128i v1 = _mm_loadu_si128(reinterpret_cast<__m128i *>(arr));
15   __m128i v2 = _mm_loadu_si128(reinterpret_cast<__m128i *>(arr + 4));
16
17   COEX_VERTICAL_128i(v1, v2);                              /* Schritt 1 */
18
19   v2 = _mm_shuffle_epi32(v2, _MM_SHUFFLE(2, 3, 0, 1));  /* Schritt 2 */
20   COEX_VERTICAL_128i(v1, v2);
21
22   auto tmp = v1;                                           /* Schritt 3 */
23   v1 = SHUFFLE_TWO_VECS(v1, v2, 0b10001000);
24   v2 = SHUFFLE_TWO_VECS(tmp, v2, 0b11011101);
25   COEX_VERTICAL_128i(v1, v2);
26
27   v2 = _mm_shuffle_epi32(v2, _MM_SHUFFLE(0, 1, 2, 3));  /* Schritt 4 */
28   COEX_VERTICAL_128i(v1, v2);
29
30   tmp = v1;                                                /* Schritt 5 */
31   v1 = SHUFFLE_TWO_VECS(v1, v2, 0b01000100);
32   v2 = SHUFFLE_TWO_VECS(tmp, v2, 0b11101110);
33   COEX_VERTICAL_128i(v1, v2);
34
35   tmp = v1;                                                /* Schritt 6 */
36   v1 = SHUFFLE_TWO_VECS(v1, v2, 0b10001000);
37   v2 = SHUFFLE_TWO_VECS(tmp, v2, 0b11011101);
38   COEX_VERTICAL_128i(v1, v2);
39
40   /* Reihenfolge wiederherstellen */
41   tmp = _mm_shuffle_epi32(v1, _MM_SHUFFLE(2, 3, 0, 1));
42   auto tmp2 = _mm_shuffle_epi32(v2, _MM_SHUFFLE(2, 3, 0, 1));
43   v2 = _mm_blend_epi32(tmp, v2, 0b00001010);
44   v1 = _mm_blend_epi32(v1, tmp2, 0b00001010);
45
46   /* speichere Vektoren */
47   _mm_storeu_si128(reinterpret_cast<__m128i* >(arr), v1);
48   _mm_storeu_si128(reinterpret_cast<__m128i*>(arr + 4), v2);
49   for (int i = 0; i < 4; ++i) printf("%d ", arr[i]);      /* Ausgabe */
50   puts("");
51   for (int i = 4; i < 8; ++i) printf("%d ", arr[i]);}
```

Quelltext A.4: Bitonic Mergesort für 16 Elemente optimiert

```
1 #define COEX_VERTICAL(a, b){  /* berechne acht Vergleichsmodule */        \
2   __m256i c = a; a = _mm256_min_epi32(a, b); b = _mm256_max_epi32(c, b);}
3
```

```
4  /* shuffle 2 Vektoren, Instruktion für int fehlt, deshalb mit float */
5  #define SHUFFLE_2_VECS(a, b, mask)                                    \
6      reinterpret_cast<__m256i>(_mm256_shuffle_ps(                      \
7          reinterpret_cast<__m256>(a), reinterpret_cast<__m256>(b), mask));
8
9  /* optimiertes Sortiernetzwerk für zwei Vektoren, d. h. 16 int */
10 inline void sort_16(__m256i &v1, __m256i &v2){
11   COEX_VERTICAL(v1, v2);                                  /* Schritt 1 */
12
13   v2 = _mm256_shuffle_epi32(v2, _MM_SHUFFLE(2, 3, 0, 1)); /* Schritt 2 */
14   COEX_VERTICAL(v1, v2);
15
16   auto tmp = v1;                                          /* Schritt  3 */
17   v1 = SHUFFLE_2_VECS(v1, v2, 0b10001000);
18   v2 = SHUFFLE_2_VECS(tmp, v2, 0b11011101);
19   COEX_VERTICAL(v1, v2);
20
21   v2 = _mm256_shuffle_epi32(v2, _MM_SHUFFLE(0, 1, 2, 3)); /* Schritt  4 */
22   COEX_VERTICAL(v1, v2);
23
24   tmp = v1;                                               /* Schritt  5 */
25   v1 = SHUFFLE_2_VECS(v1, v2, 0b01000100);
26   v2 = SHUFFLE_2_VECS(tmp, v2, 0b11101110);
27   COEX_VERTICAL(v1, v2);
28
29   tmp = v1;                                               /* Schritt  6 */
30   v1 = SHUFFLE_2_VECS(v1, v2, 0b11011000);
31   v2 = SHUFFLE_2_VECS(tmp, v2, 0b10001101);
32   COEX_VERTICAL(v1, v2);
33
34   v2 = _mm256_permutevar8x32_epi32(v2, _mm256_setr_epi32(7,6,5,4,3,2,1,0));
35   COEX_VERTICAL(v1, v2);                                  /* Schritt  7 */
36
37   tmp = v1;                                               /* Schritt  8 */
38   v1 = SHUFFLE_2_VECS(v1, v2, 0b11011000);
39   v2 = SHUFFLE_2_VECS(tmp, v2, 0b10001101);
40   COEX_VERTICAL(v1, v2);
41
42   tmp = v1;                                               /* Schritt  9 */
43   v1 = SHUFFLE_2_VECS(v1, v2, 0b11011000);
44   v2 = SHUFFLE_2_VECS(tmp, v2, 0b10001101);
45   COEX_VERTICAL(v1, v2);
46
47   /* permutieren, um Reihenfolge einfacher wiederherzustellen */
48   v1 = _mm256_permutevar8x32_epi32(v1, _mm256_setr_epi32(0,4,1,5,6,2,7,3));
```

```
49   v2 = _mm256_permutevar8x32_epi32(v2, _mm256_setr_epi32(0,4,1,5,6,2,7,3));
50
51   tmp = v1;                                                  /* Schritt  10 */
52   v1 = SHUFFLE_2_VECS(v1, v2, 0b10001000);
53   v2 = SHUFFLE_2_VECS(tmp, v2, 0b11011101);
54   COEX_VERTICAL(v1, v2);
55
56   /* Reihenfolge wiederherstellen */
57   auto b2 = _mm256_shuffle_epi32(v2,0b10110001);
58   auto b1 = _mm256_shuffle_epi32(v1,0b10110001);
59   v1 = _mm256_blend_epi32(v1, b2, 0b10101010);
60   v2 = _mm256_blend_epi32(b1, v2, 0b10101010);}
```

Quelltext A.5: Shellsort zur Sortierung von 8 Spalten

```
1  #define COEX_VERTICAL(a, b){  /* berechne acht Vergleichsmodule */        \
2    __m256i c = a; a = _mm256_min_epi32(a, b); b = _mm256_max_epi32(c, b);}
3
4  /* wenn mindestens ein Wert in v1 größer als in v2 ist, gibt true zurück */
5  inline bool is_greater(__m256i& v1, __m256i& v2){
6    return 0 != _mm256_movemask_epi8(_mm256_cmpgt_epi32(v1, v2));}
7
8  /* Funktion sortiert spaltenweise die Werte der Vektoren
9   * Eingabe ist Array aus Vektoren (vecs) und Anzahl der Vektoren (N) */
10 inline void shell_sort(__m256i *vecs, const int N){
11   int i, j; __m256i temp;
12   for (auto gap : {1750, 701, 301, 132, 57, 23, 10, 4, 1}){
13     for (i = gap; i < N; i += 1){
14       temp = vecs[i];
15       for (j = i; j >= gap && is_greater(vecs[j - gap], temp); j -= gap){
16         COEX_VERTICAL(vecs[j - gap], vecs[j])}
17       COEX_VERTICAL(vecs[j], temp)}}}
```

Quelltext A.6: Spaltenweise Sortierung von 16 Elementen und Transposition

```
1  #include <cstdio>
2  #include <immintrin.h>
3  #include <stdlib.h>
4
5  #define COEX_VERTICAL(a, b){  /* berechne acht Vergleichsmodule */        \
6    __m256i c = a; a = _mm256_min_epi32(a, b); b = _mm256_max_epi32(c, b);}
7
8  /* transponiere 8 mal 8 Matrix mit Integern */
9  inline void transpose_8x8_int(__m256i *vecs){
10   __m256 *v = reinterpret_cast<__m256 *>(vecs);
11   __m256 a = _mm256_unpacklo_ps(v[0], v[1]);
```

```
12    __m256 b = _mm256_unpackhi_ps(v[0], v[1]);
13    __m256 c = _mm256_unpacklo_ps(v[2], v[3]);
14    __m256 d = _mm256_unpackhi_ps(v[2], v[3]);
15    __m256 e = _mm256_unpacklo_ps(v[4], v[5]);
16    __m256 f = _mm256_unpackhi_ps(v[4], v[5]);
17    __m256 g = _mm256_unpacklo_ps(v[6], v[7]);
18    __m256 h = _mm256_unpackhi_ps(v[6], v[7]);
19    auto tmp = _mm256_shuffle_ps(a, c, 0x4E);
20    a = _mm256_blend_ps(a, tmp, 0xCC);
21    c = _mm256_blend_ps(c, tmp, 0x33);
22    tmp = _mm256_shuffle_ps(b, d, 0x4E);
23    b = _mm256_blend_ps(b, tmp, 0xCC);
24    d = _mm256_blend_ps(d, tmp, 0x33);
25    tmp = _mm256_shuffle_ps(e, g, 0x4E);
26    e = _mm256_blend_ps(e, tmp, 0xCC);
27    g = _mm256_blend_ps(g, tmp, 0x33);
28    tmp = _mm256_shuffle_ps(f, h, 0x4E);
29    f = _mm256_blend_ps(f, tmp, 0xCC);
30    h = _mm256_blend_ps(h, tmp, 0x33);
31    v[0] = _mm256_permute2f128_ps(a, e, 0x20);
32    v[1] = _mm256_permute2f128_ps(c, g, 0x20);
33    v[2] = _mm256_permute2f128_ps(b, f, 0x20);
34    v[3] = _mm256_permute2f128_ps(d, h, 0x20);
35    v[4] = _mm256_permute2f128_ps(a, e, 0x31);
36    v[5] = _mm256_permute2f128_ps(c, g, 0x31);
37    v[6] = _mm256_permute2f128_ps(b, f, 0x31);
38    v[7] = _mm256_permute2f128_ps(d, h, 0x31);}
39
40  #define CV(a, b) COEX_VERTICAL(a, b) /* Alias für weniger Code */
41
42  /* sortiere in 8 Spalten jeweils 16 int mit 60 Modulen */
43  inline void sort_16_int_vertical(__m256i* vecs){
44    CV(vecs[0], vecs[1]); CV(vecs[2], vecs[3]);  /* Schritt 1 */
45    CV(vecs[4], vecs[5]); CV(vecs[6], vecs[7]);
46    CV(vecs[8], vecs[9]); CV(vecs[10], vecs[11])
47    CV(vecs[12], vecs[13]); CV(vecs[14], vecs[15])
48    CV(vecs[0], vecs[2]); CV(vecs[1], vecs[3]);  /* Schritt 2 */
49    CV(vecs[4], vecs[6]); CV(vecs[5], vecs[7]);
50    CV(vecs[8], vecs[10]); CV(vecs[9], vecs[11]);
51    CV(vecs[12], vecs[14]); CV(vecs[13], vecs[15]);
52    CV(vecs[0], vecs[4]); CV(vecs[1], vecs[5]);  /* Schritt 3 */
53    CV(vecs[2], vecs[6]); CV(vecs[3], vecs[7]);
54    CV(vecs[8], vecs[12]); CV(vecs[9], vecs[13]);
55    CV(vecs[10], vecs[14]); CV(vecs[11], vecs[15]);
56    CV(vecs[0], vecs[8]); CV(vecs[1], vecs[9])   /* Schritt 4 */
```

```
57    CV(vecs[2], vecs[10]); CV(vecs[3], vecs[11])
58    CV(vecs[4], vecs[12]); CV(vecs[5], vecs[13])
59    CV(vecs[6], vecs[14]); CV(vecs[7], vecs[15])
60    CV(vecs[5], vecs[10]); CV(vecs[6], vecs[9]); /* Schritt 5 */
61    CV(vecs[3], vecs[12]); CV(vecs[7], vecs[11]);
62    CV(vecs[13], vecs[14]); CV(vecs[4], vecs[8]);
63    CV(vecs[1], vecs[2]);
64    CV(vecs[1], vecs[4]); CV(vecs[7], vecs[13]); /* Schritt 6 */
65    CV(vecs[2], vecs[8]); CV(vecs[11], vecs[14]);
66    CV(vecs[2], vecs[4]); CV(vecs[5], vecs[6]);  /* Schritt 7 */
67    CV(vecs[9], vecs[10]); CV(vecs[11], vecs[13]);
68    CV(vecs[3], vecs[8]); CV(vecs[7], vecs[12]);
69    CV(vecs[3], vecs[5]); CV(vecs[6], vecs[8]);  /* Schritt 8 */
70    CV(vecs[7], vecs[9]); CV(vecs[10], vecs[12]);
71    CV(vecs[3], vecs[4]); CV(vecs[5], vecs[6]);  /* Schritt 9 */
72    CV(vecs[7], vecs[8]); CV(vecs[9], vecs[10]);
73    CV(vecs[11], vecs[12]);
74    CV(vecs[6], vecs[7]); CV(vecs[8], vecs[9]); /* Schritt 10 */}
75
76  #define SWAP(a, b) {auto vec_tmp = a; a = b; b = vec_tmp;}
77
78  int main(){
79    __m256i vecs[16];     /* initialisiere Vektoren mit Zufallszahlen */
80    int* arr = reinterpret_cast<int*>(vecs);
81    for (int i = 0; i < 128; ++i) arr[i] = rand() % 100;
82
83    sort_16_int_vertical(vecs);              /* sortiere spaltenweise */
84    transpose_8x8_int(vecs);          /* transponiere erste  64 Werte */
85    transpose_8x8_int(vecs + 8);     /* transponiere letzte  64 Werte */
86    /* 16 sortierte Elemente in jeder Zeile erzeugen */
87    SWAP(vecs[1], vecs[8]); SWAP(vecs[3], vecs[10]);
88    SWAP(vecs[5], vecs[12]); SWAP(vecs[7], vecs[14]);
89
90    for (int i = 0; i < 128; i+=16){          /* zeilenweise Ausgabe */
91      for (int j = 0; j < 16; ++j){
92        printf("%d ", arr[i + j]);}  /* einzelne Zeilen sind sortiert */
93      puts("");}}
```

Quelltext A.7: Modulbasierte Sortierung von N Vektoren ohne Transposition

```
1  /* Hilfsfunktion zum spaltenweisen Sortieren und Mergen */
2  inline void oddeven_mergesort(__m256i *vecs, int N, const int s = 2) {
3    for (int t = s; t < N * 2; t *= 2) {
4      for (int l = 0; l < N; l += t) {
5        const int bound = std::min(l + t / 2, N - t / 2);
6        for (int i = l; i < bound; ++i) {
```

```
7         COEX_VERTICAL(vecs[i], vecs[i + t / 2]); }
8       for (int j = t / 4; j > 0; j /= 2) {
9         for (int i = j; i < t - j * 2; i += 2 * j) {
10          const int bound = std::min(i + l + j, N - j);
11          for (int k = i + l; k < bound; ++k) {
12            COEX_VERTICAL(vecs[k], vecs[k + j]); }}}}}
13
14 /* Hilfsfunktion zur Ermittlung der nächsten Zweierpotenz von x */
15 inline int next_power_of_two(int x) {
16   x--; x |= x >> 1; x |= x >> 2; x |= x >> 4;
17   x |= x >> 8; x |= x >> 16; x++;
18   return x;}
19
20 /* N Vektoren sortieren, d. h. N * 8 int */
21 void no_transpose_sort(__m256i *vecs, const int N) {
22   /* Phase 1: Sortiere Spalten */
23   /* spaltenweise Sechzehnergruppen sortieren */
24   for (int i = 0; i < N - N % 16; i += 16){sort_16_int_vertical(vecs + i);}
25   /* sortiere letzte Vektoren mit Odd-Even Mergesort, falls N % 16 != 0 */
26   oddeven_mergesort(vecs + N - N % 16, N % 16);
27   /* Merge in Spalten Sechzehnergruppen */
28   oddeven_mergesort(vecs, N, 16);
29
30   /* Phase 2: Mit Bitonic Merge die sortierten Spalten mergen */
31   const int npot = next_power_of_two(N);
32   for (int m = npot/2; m > 0; m/=2) {
33     for (int i = m; i < N; i += 2 * m) {
34       const int bound = std::min(N, i + m);
35       for (int j = i, k=1; j < bound; ++j, ++k) {
36         vecs[j] = _mm256_shuffle_epi32(vecs[j], _MM_SHUFFLE(2,3,0,1));
37         COEX_VERTICAL(vecs[i - k], vecs[j]) }}}
38   for (int l = 0; l < N; ++l) {
39     COEX_SHUFFLE(vecs[l], 1, 0, 3, 2, 5, 4, 7, 6, ASC); }
40   for (int m = npot / 2; m > 0; m/=2) {
41     for (int i = m; i < N; i += 2 * m) {
42       const int bound = std::min(N, i + m);
43       for (int j = i, k=1; j < bound; ++j, ++k) {
44         vecs[j] = _mm256_shuffle_epi32(vecs[j], _MM_SHUFFLE(0,1,2,3));
45         COEX_VERTICAL(vecs[i - k], vecs[j]) }}}
46   for (int l = 0; l < N; ++l) {
47     COEX_SHUFFLE(vecs[l], 3, 2, 1, 0, 7, 6, 5, 4, ASC);
48     COEX_SHUFFLE(vecs[l], 1, 0, 3, 2, 5, 4, 7, 6, ASC); }
49   for (int m = npot / 2; m > 0; m /= 2) {
50     for (int i = m; i < N; i += 2 * m) {
51       const int bound = std::min(N, i + m);
```

```
52        for (int j = i, k = 1; j < bound; ++j, ++k) {
53          REVERSE_VEC(vecs[j]);
54          COEX_VERTICAL(vecs[i - k], vecs[j]) }}}
55    for (int l = 0; l < N; ++l) {
56      COEX_PERMUTE(vecs[l], 7, 6, 5, 4, 3, 2, 1, 0, ASC);
57      COEX_SHUFFLE(vecs[l], 2, 3, 0, 1, 6, 7, 4, 5, ASC);
58      COEX_SHUFFLE(vecs[l], 1, 0, 3, 2, 5, 4, 7, 6, ASC); }}
```

Quelltext A.8: Bitonic Merge optimiert

```
1  /* N Vektoren mergen, N ist Anzahl der Vektoren, N % 2 == 0 und N > 0
2   * s = 2 bedeutet, dass jeweils zwei Vektoren bereits sortiert sind */
3  inline void bitonic_merge_16(__m256i *vecs, const int N, const int s = 2) {
4    for (int t = s * 2; t < 2 * N; t *= 2) {
5      for (int l = 0; l < N; l += t) {
6        for (int j = std::max(l + t - N, 0); j < t/2 ; j += 2) {
7          REVERSE_VEC(vecs[l + t - 1 - j]);
8          REVERSE_VEC(vecs[l + t - 2 - j]);
9          COEX_VERTICAL(vecs[l + j], vecs[l + t - 1 - j]);
10         COEX_VERTICAL(vecs[l + j + 1], vecs[l + t - 2 - j]); }}
11     for (int m = t / 2; m > 4; m /= 2) {
12       for (int k = 0; k < N - m / 2; k += m) {
13         const int bound = std::min((k + m / 2), N - (m / 2));
14         for (int j = k; j < bound; j += 2) {
15           COEX_VERTICAL(vecs[j], vecs[m / 2 + j]);
16           COEX_VERTICAL(vecs[j + 1], vecs[m / 2 + j + 1]); }}}
17     for (int j = 0; j < N - 2; j += 4) {
18       COEX_VERTICAL(vecs[j], vecs[j + 2]);
19       COEX_VERTICAL(vecs[j + 1], vecs[j + 3]); }
20     for (int j = 0; j < N; j += 2) {
21       COEX_VERTICAL(vecs[j], vecs[j + 1]); }
22     for (int i = 0; i < N; i += 2) {
23       COEX_PERMUTE(vecs[i], 4, 5, 6, 7, 0, 1, 2, 3, ASC);
24       COEX_PERMUTE(vecs[i + 1], 4, 5, 6, 7, 0, 1, 2, 3, ASC);
25       auto tmp = vecs[i]; /* 8 Module gleichzeitig mit COEX_VERTICAL */
26       vecs[i] = _mm256_unpacklo_epi32(vecs[i], vecs[i + 1]);
27       vecs[i + 1] = _mm256_unpackhi_epi32(tmp, vecs[i + 1]);
28       COEX_VERTICAL(vecs[i], vecs[i + 1]);
29       tmp = vecs[i];
30       vecs[i] = _mm256_unpacklo_epi32(vecs[i], vecs[i + 1]);
31       vecs[i + 1] = _mm256_unpackhi_epi32(tmp, vecs[i + 1]);
32       COEX_VERTICAL(vecs[i], vecs[i + 1]);
33       tmp = vecs[i];
34       vecs[i] = _mm256_unpacklo_epi32(vecs[i], vecs[i + 1]);
35       vecs[i + 1] = _mm256_unpackhi_epi32(tmp, vecs[i + 1]); }}}
```

Quelltext A.9: Vektorisierter Mergesort optimiert

```
#define UNPACK(vecs) {            /* statt shuffle, Vektoren entpacken */ \
  auto tmp = vecs[0]; auto tmp2 = vecs[2];                                 \
  vecs[0] = _mm256_unpacklo_epi32(vecs[0], vecs[1]);                       \
  vecs[1] = _mm256_unpackhi_epi32(tmp, vecs[1]);                           \
  vecs[2] = _mm256_unpacklo_epi32(vecs[2], vecs[3]);                       \
  vecs[3] = _mm256_unpackhi_epi32(tmp2, vecs[3]); }

#define CV(a, b) COEX_VERTICAL(a, b) /* Aliase für weniger Code */
#define CP(vec) COEX_PERMUTE(vec, 4, 5, 6, 7, 0, 1, 2, 3, ASC);

/* 4 sortierte Vektoren in v_min mit 4 sortierten Vektoren in v_max mit
 * Bitonic Merge mergen, die vier Vektoren von v_min in c speichern */
inline void merge_2x4_vecs(__m256i *v_min, __m256i *v_max, __m256i *c) {
  REVERSE_VEC(v_max[0]); REVERSE_VEC(v_max[1]); REVERSE_VEC(v_max[2]);
  REVERSE_VEC(v_max[3]); CV(v_min[0], v_max[3]); CV(v_min[1], v_max[2]);
  CV(v_min[2], v_max[1]); CV(v_min[3], v_max[0]); CV(v_min[0], v_min[2]);
  CV(v_min[1], v_min[3]); CV(v_min[0], v_min[1]); CV(v_min[2], v_min[3]);
  CP(v_min[0]); CP(v_min[1]); CP(v_min[2]); CP(v_min[3]); UNPACK(v_min);
  CV(v_min[0], v_min[1]); CV(v_min[2], v_min[3]); UNPACK(v_min);
  CV(v_min[0], v_min[1]); CV(v_min[2], v_min[3]); UNPACK(v_min);
  c[0] = v_min[0]; c[1] = v_min[1]; c[2] = v_min[2]; c[3] = v_min[3];
  CV(v_max[0], v_max[2]); CV(v_max[1], v_max[3]); CV(v_max[0], v_max[1]);
  CV(v_max[2], v_max[3]); CP(v_max[0]); CP(v_max[1]); CP(v_max[2]);
  CP(v_max[3]); UNPACK(v_max); CV(v_max[0], v_max[1]);
  CV(v_max[2], v_max[3]); UNPACK(v_max); CV(v_max[0], v_max[1]);
  CV(v_max[2], v_max[3]); UNPACK(v_max); }

inline void merge_optimized(__m256i *a, __m256i *b, __m256i *c,
                    const int a_size, const int b_size, const int c_size) {
  __m256i *v_min = a, *v_max = b;      /* v_min und v_max initialisieren */
  int idx_a = 4, idx_b = 4;
  for (int i = 0; i < c_size - 4; i += 4) {
    merge_2x4_vecs(v_min, v_max, &c[i]);       /* v_min und v_max mergen */
    if (idx_a == a_size) { v_min = &b[idx_b]; b += 4; }          //a fertig
    else if (idx_b == b_size) { v_min = &a[idx_a]; idx_a += 4; } //b fertig
    else {
      if (_mm256_extract_epi32(a[idx_a], 0) <
          _mm256_extract_epi32(b[idx_b], 0)) {
               v_min = &a[idx_a]; idx_a += 4; }
      else { v_min = &b[idx_b]; idx_b += 4; }}}
  for (int j = 0; j < 4; ++j) { c[c_size - 4 + j] = v_max[j]; }}

inline void merge_sort_optimized(__m256i *, __m256i *, int);

```

```
/* Hilfsfunktion vermeidet Kopieren der Werte von c nach vecs */
inline void merge_sort_helper(__m256i *vecs, __m256i *c, const int N) {
  const int size_a = N / 2 - (N / 2) % 4;          /* Länge Teilfolge a */
  const int size_b = N - size_a;                   /* Länge Teilfolge b */
  merge_sort_optimized(vecs, c, size_a);         /* Teilfolgen sortieren */
  merge_sort_optimized(vecs + size_a, c + size_a, size_b);
  merge_optimized(vecs, vecs + size_a, c, size_a, size_b, N); }

/* N Vektoren sortieren, es gilt N % 4 == 0, c ist zusätzlicher Speicher */
inline void merge_sort_optimized(__m256i *vecs, __m256i *c, const int N) {
  /* falls weniger als 1025 Vektoren mit vektorisiertem Sortiernetzwerk */
  if (N < 1025) { sort_int_sorting_network_aligned(vecs, N); return; }
  const int size_a = N / 2 - (N / 2) % 4;          /* Länge Teilfolge a */
  const int size_b = N - size_a;                   /* Länge Teilfolge b */
  merge_sort_helper(vecs, c, size_a);            /* Teilfolgen sortieren */
  merge_sort_helper(vecs + size_a, c + size_a, size_b);
  /* Zeiger vecs und c vertauschen vermeidet Kopieren von c nach vecs */
  merge_optimized(c, c + size_a, vecs, size_a, size_b, N); } // mergen
```

Quelltext A.10: Naiv implementierter Quicksort

```
int partition(int *array, int left, int right, int pivot) {
  for (int i = left; i < right; ++i) {
    /* Werte kleiner als Pivot ins linke Teilarray bewegen */
    if (array[i] < pivot) { swap(array[i], array[left++]); }}
  return left; }

void quicksort(int *arr, int left, int right) {
  if (left < right) { /* falls mind. 2 Elemente im Array */
    /* Pivot-Wert in der Mitte des Arrays */
    int pivot_idx = ((right - left) / 2) + left;
    /* Pivot-Wert ans Ende des Arrays setzen */
    swap(arr[pivot_idx], arr[right]);
    int bound = partition(arr, left, right, arr[right]);
    /* Pivot-Wert an sortierte Position im Array setzen */
    swap(arr[bound], arr[right]);
    quicksort(arr, left, bound - 1); /* linkes Teilarray */
    quicksort(arr, bound + 1, right); }} /* rechtes Teilarray */

void qs_sort(int *array, int length) { quicksort(array, 0, length - 1); }
```

Quelltext A.11: Generierung von Permutations-Masken für Quicksort

```
#include <iostream>
#include <vector>
#include <string>
```

```

using namespace std;

string generate_permutation_masks(uint32_t mask) {
  string permutation_masks = "_mm256_setr_epi32(";
  vector<int> perm_gt{};
  vector<int> perm_all{};
  for (int i = 0; i < 8; ++i) {
    if(mask & 1){ perm_gt.push_back(i); }
    else { perm_all.push_back(i); }
    mask >>= 1; }
  perm_all.insert(end(perm_all), begin(perm_gt), end(perm_gt));
  for (int i = 0; i < 8; ++i) {
    permutation_masks += to_string(perm_all[i]);
    if(i != 7){ permutation_masks += ", "; }}
  return permutation_masks + ")"; }

int main() {
  string code = "const __m256i permutation_masks[256] = {\n";
  for (uint32_t i = 0; i < 256; ++i) {
    code += generate_permutation_masks(i);
    if(i != 255){
      code += ", \n"; }}
  code += "};\n";
  cout << code << endl; }
```

Quelltext A.12: Berechnung von Minimum und Maximum im Vektor

```
inline int calc_min(__m256i vec) { /* Minimum aus 8 int */
  auto perm_mask = _mm256_setr_epi32(7, 6, 5, 4, 3, 2, 1, 0);
  vec = _mm256_min_epi32(vec, _mm256_permutevar8x32_epi32(vec, perm_mask));
  vec = _mm256_min_epi32(vec, _mm256_shuffle_epi32(vec, 0b10110001));
  vec = _mm256_min_epi32(vec, _mm256_shuffle_epi32(vec, 0b01001110));
  return _mm256_extract_epi32(vec, 0); }

inline int calc_max(__m256i vec) { /* Maximum aus 8 int */
  auto perm_mask = _mm256_setr_epi32(7, 6, 5, 4, 3, 2, 1, 0);
  vec = _mm256_max_epi32(vec, _mm256_permutevar8x32_epi32(vec, perm_mask));
  vec = _mm256_max_epi32(vec, _mm256_shuffle_epi32(vec, 0b10110001));
  vec = _mm256_max_epi32(vec, _mm256_shuffle_epi32(vec, 0b01001110));
  return _mm256_extract_epi32(vec, 0); }
```

Quelltext A.13: Optimierte vektorisierte Partitionierung des Arrays

```
inline int partition_vectorized_64(int *arr, int left, int right,
                                   int pivot, int &smallest, int &biggest) {
```

```
3   if (right - left < 129) { /* nicht optimieren, wenn weniger als 129 Elemente */
4     return partition_vectorized_8(arr, left, right, pivot, smallest, biggest); }
5
6   for (int i = (right - left) % 8; i > 0; --i) { /* Array kürzen */
7     smallest = min(smallest, arr[left]); biggest = max(biggest, arr[left]);
8     if (arr[left] > pivot) { swap(arr[left], arr[--right]); }
9     else { ++left; }}
10
11  auto pivot_vec = _mm256_set1_epi32(pivot); /* Vektor befüllt mit Pivot */
12  auto sv = _mm256_set1_epi32(smallest); /* Vektor für smallest */
13  auto bv = _mm256_set1_epi32(biggest); /* Vektor für biggest */
14
15  /* vektorweise Array kürzen, bis Anzahl der Elemente ohne Rest durch 64 teilbar */
16  for (int i = ((right - left) % 64) / 8; i > 0; --i) {
17    __m256i vec_L = LOAD_VEC(arr + left);
18    /* wie in Funktion partition_vec */
19    __m256i compared = _mm256_cmpgt_epi32(vec_L, pivot_vec);
20    sv = _mm256_min_epi32(vec_L, sv); bv = _mm256_max_epi32(vec_L, bv);
21    int mm = _mm256_movemask_ps(reinterpret_cast<__m256>(compared));
22    int amount_gt_pivot = _mm_popcnt_u32((mm));
23    __m256i permuted = _mm256_permutevar8x32_epi32(vec_L, permutation_masks[mm]);
24
25    /* dies ist eine langsamere Möglichkeit, vektorisiert ein Array zu partitionieren */
26    __m256i blend_mask = _mm256_cmpgt_epi32(permuted, pivot_vec);
27    __m256i vec_R = LOAD_VEC(arr + right - 8);
28    __m256i vec_L_new = _mm256_blendv_epi8(permuted, vec_R, blend_mask);
29    __m256i vec_R_new = _mm256_blendv_epi8(vec_R, permuted, blend_mask);
30    STORE_VEC(arr + left, vec_L_new); STORE_VEC(arr + right - 8, vec_R_new);
31    left += (8 - amount_gt_pivot); right -= amount_gt_pivot; }
32
33  /* jeweils 8 Vektoren von beiden Seiten des Arrays zwischenspeichern */
34  auto vec_left = LOAD_VEC(arr + left), vec_left2 = LOAD_VEC(arr + left + 8);
35  auto vec_left3 = LOAD_VEC(arr + left + 16), vec_left4 = LOAD_VEC(arr + left + 24);
36  auto vec_left5 = LOAD_VEC(arr + left + 32), vec_left6 = LOAD_VEC(arr + left + 40);
37  auto vec_left7 = LOAD_VEC(arr + left + 48), vec_left8 = LOAD_VEC(arr + left + 56);
38  auto vec_right = LOAD_VEC(arr + (right - 64)), vec_right2 = LOAD_VEC(arr + (right - 56));
39  auto vec_right3 = LOAD_VEC(arr + (right - 48)), vec_right4 = LOAD_VEC(arr + (right - 40));
40  auto vec_right5 = LOAD_VEC(arr + (right - 32)), vec_right6 = LOAD_VEC(arr + (right - 24));
41  auto vec_right7 = LOAD_VEC(arr + (right - 16)), vec_right8 = LOAD_VEC(arr + (right - 8));
42
43  /* Positionen, an denen die Vektoren gespeichert werden */
44  int r_store = right - 64; /* rechte Position zum Speichern */
45  int l_store = left; /* linke Position zum Speichern */
46  /* Positionen, an denen Vektoren geladen werden */
47  left += 64; /* erhöhen, da erste 64 Elemente zwischengespeichert */
48  right -= 64; /* verringern, da letzte 64 Elemente zwischengespeichert */
49
50  while (right - left != 0) { /* 64 Elemente pro Iteration partitionieren */
51    __m256i curr_vec, curr_vec2, curr_vec3, curr_vec4, curr_vec5, curr_vec6, curr_vec7,
      curr_vec8;
52
53    /* wenn weniger Elemente auf rechter Seite des Arrays gespeichert, dann
54     * nächste 8 Vektoren von rechter Seite, sonst von linker Seite laden */
55    if ((r_store + 64) - right < left - l_store) {
56      right -= 64;
57      curr_vec = LOAD_VEC(arr + right); curr_vec2 = LOAD_VEC(arr + right + 8);
58      curr_vec3 = LOAD_VEC(arr + right + 16); curr_vec4 = LOAD_VEC(arr + right + 24);
```

```
59        curr_vec5 = LOAD_VEC(arr + right + 32); curr_vec6 = LOAD_VEC(arr + right + 40);
60        curr_vec7 = LOAD_VEC(arr + right + 48); curr_vec8 = LOAD_VEC(arr + right + 56); }
61      else {
62        curr_vec = LOAD_VEC(arr + left); curr_vec2 = LOAD_VEC(arr + left + 8);
63        curr_vec3 = LOAD_VEC(arr + left + 16); curr_vec4 = LOAD_VEC(arr + left + 24);
64        curr_vec5 = LOAD_VEC(arr + left + 32); curr_vec6 = LOAD_VEC(arr + left + 40);
65        curr_vec7 = LOAD_VEC(arr + left + 48); curr_vec8 = LOAD_VEC(arr + left + 56);
66        left += 64; }
67
68      /* 8 Vektoren partitionieren und auf beiden Seiten des Arrays speichern */
69      int amount_gt_pivot = partition_vec(curr_vec, pivot_vec, sv, bv);
70      int amount_gt_pivot2 = partition_vec(curr_vec2, pivot_vec, sv, bv);
71      int amount_gt_pivot3 = partition_vec(curr_vec3, pivot_vec, sv, bv);
72      int amount_gt_pivot4 = partition_vec(curr_vec4, pivot_vec, sv, bv);
73      int amount_gt_pivot5 = partition_vec(curr_vec5, pivot_vec, sv, bv);
74      int amount_gt_pivot6 = partition_vec(curr_vec6, pivot_vec, sv, bv);
75      int amount_gt_pivot7 = partition_vec(curr_vec7, pivot_vec, sv, bv);
76      int amount_gt_pivot8 = partition_vec(curr_vec8, pivot_vec, sv, bv);
77
78      STORE_VEC(arr + l_store, curr_vec); l_store += (8 - amount_gt_pivot);
79      STORE_VEC(arr + l_store, curr_vec2); l_store += (8 - amount_gt_pivot2);
80      STORE_VEC(arr + l_store, curr_vec3); l_store += (8 - amount_gt_pivot3);
81      STORE_VEC(arr + l_store, curr_vec4); l_store += (8 - amount_gt_pivot4);
82      STORE_VEC(arr + l_store, curr_vec5); l_store += (8 - amount_gt_pivot5);
83      STORE_VEC(arr + l_store, curr_vec6); l_store += (8 - amount_gt_pivot6);
84      STORE_VEC(arr + l_store, curr_vec7); l_store += (8 - amount_gt_pivot7);
85      STORE_VEC(arr + l_store, curr_vec8); l_store += (8 - amount_gt_pivot8);
86
87      STORE_VEC(arr + r_store + 56, curr_vec); r_store -= amount_gt_pivot;
88      STORE_VEC(arr + r_store + 56, curr_vec2); r_store -= amount_gt_pivot2;
89      STORE_VEC(arr + r_store + 56, curr_vec3); r_store -= amount_gt_pivot3;
90      STORE_VEC(arr + r_store + 56, curr_vec4); r_store -= amount_gt_pivot4;
91      STORE_VEC(arr + r_store + 56, curr_vec5); r_store -= amount_gt_pivot5;
92      STORE_VEC(arr + r_store + 56, curr_vec6); r_store -= amount_gt_pivot6;
93      STORE_VEC(arr + r_store + 56, curr_vec7); r_store -= amount_gt_pivot7;
94      STORE_VEC(arr + r_store + 56, curr_vec8); r_store -= amount_gt_pivot8; }
95
96    /* 8 Vektoren die, von der linken Seite des Arrays stammen, partitionieren und speichern */
97    int amount_gt_pivot = partition_vec(vec_left, pivot_vec, sv, bv);
98    int amount_gt_pivot2 = partition_vec(vec_left2, pivot_vec, sv, bv);
99    int amount_gt_pivot3 = partition_vec(vec_left3, pivot_vec, sv, bv);
100   int amount_gt_pivot4 = partition_vec(vec_left4, pivot_vec, sv, bv);
101   int amount_gt_pivot5 = partition_vec(vec_left5, pivot_vec, sv, bv);
102   int amount_gt_pivot6 = partition_vec(vec_left6, pivot_vec, sv, bv);
103   int amount_gt_pivot7 = partition_vec(vec_left7, pivot_vec, sv, bv);
104   int amount_gt_pivot8 = partition_vec(vec_left8, pivot_vec, sv, bv);
105
106   STORE_VEC(arr + l_store, vec_left); l_store += (8 - amount_gt_pivot);
107   STORE_VEC(arr + l_store, vec_left2); l_store += (8 - amount_gt_pivot2);
108   STORE_VEC(arr + l_store, vec_left3); l_store += (8 - amount_gt_pivot3);
109   STORE_VEC(arr + l_store, vec_left4); l_store += (8 - amount_gt_pivot4);
110   STORE_VEC(arr + l_store, vec_left5); l_store += (8 - amount_gt_pivot5);
111   STORE_VEC(arr + l_store, vec_left6); l_store += (8 - amount_gt_pivot6);
112   STORE_VEC(arr + l_store, vec_left7); l_store += (8 - amount_gt_pivot7);
113   STORE_VEC(arr + l_store, vec_left8); l_store += (8 - amount_gt_pivot8);
114
115   STORE_VEC(arr + r_store + 56, vec_left); r_store -= amount_gt_pivot;
```

```
116  STORE_VEC(arr + r_store + 56, vec_left2); r_store -= amount_gt_pivot2;
117  STORE_VEC(arr + r_store + 56, vec_left3); r_store -= amount_gt_pivot3;
118  STORE_VEC(arr + r_store + 56, vec_left4); r_store -= amount_gt_pivot4;
119  STORE_VEC(arr + r_store + 56, vec_left5); r_store -= amount_gt_pivot5;
120  STORE_VEC(arr + r_store + 56, vec_left6); r_store -= amount_gt_pivot6;
121  STORE_VEC(arr + r_store + 56, vec_left7); r_store -= amount_gt_pivot7;
122  STORE_VEC(arr + r_store + 56, vec_left8); r_store -= amount_gt_pivot8;
123
124  /* 8 Vektoren, die von der rechten Seite des Arrays stammen, partitionieren und speichern */
125  amount_gt_pivot = partition_vec(vec_right, pivot_vec, sv, bv);
126  amount_gt_pivot2 = partition_vec(vec_right2, pivot_vec, sv, bv);
127  amount_gt_pivot3 = partition_vec(vec_right3, pivot_vec, sv, bv);
128  amount_gt_pivot4 = partition_vec(vec_right4, pivot_vec, sv, bv);
129  amount_gt_pivot5 = partition_vec(vec_right5, pivot_vec, sv, bv);
130  amount_gt_pivot6 = partition_vec(vec_right6, pivot_vec, sv, bv);
131  amount_gt_pivot7 = partition_vec(vec_right7, pivot_vec, sv, bv);
132  amount_gt_pivot8 = partition_vec(vec_right8, pivot_vec, sv, bv);
133
134  STORE_VEC(arr + l_store, vec_right); l_store += (8 - amount_gt_pivot);
135  STORE_VEC(arr + l_store, vec_right2); l_store += (8 - amount_gt_pivot2);
136  STORE_VEC(arr + l_store, vec_right3); l_store += (8 - amount_gt_pivot3);
137  STORE_VEC(arr + l_store, vec_right4); l_store += (8 - amount_gt_pivot4);
138  STORE_VEC(arr + l_store, vec_right5); l_store += (8 - amount_gt_pivot5);
139  STORE_VEC(arr + l_store, vec_right6); l_store += (8 - amount_gt_pivot6);
140  STORE_VEC(arr + l_store, vec_right7); l_store += (8 - amount_gt_pivot7);
141  STORE_VEC(arr + l_store, vec_right8); l_store += (8 - amount_gt_pivot8);
142
143  STORE_VEC(arr + r_store + 56, vec_right); r_store -= amount_gt_pivot;
144  STORE_VEC(arr + r_store + 56, vec_right2); r_store -= amount_gt_pivot2;
145  STORE_VEC(arr + r_store + 56, vec_right3); r_store -= amount_gt_pivot3;
146  STORE_VEC(arr + r_store + 56, vec_right4); r_store -= amount_gt_pivot4;
147  STORE_VEC(arr + r_store + 56, vec_right5); r_store -= amount_gt_pivot5;
148  STORE_VEC(arr + r_store + 56, vec_right6); r_store -= amount_gt_pivot6;
149  STORE_VEC(arr + r_store + 56, vec_right7); r_store -= amount_gt_pivot7;
150  STORE_VEC(arr + r_store + 56, vec_right8); r_store -= amount_gt_pivot8;
151
152  smallest = calc_min(sv); biggest = calc_max(bv);
153  return l_store; }
```

Quelltext A.14: Rekursive Funktion für vektorisierten Quicksort

```
1  inline void qs_core(int *arr, int left, int right,
2                      bool choose_avg = false, const int avg = 0) {
3    if (right - left < 513) {   /* Sortiernetzwerke für kleine Teilarrays */
4      static __m256i buffer[66];              /* Buffer (singlethreaded) */
5      /*** bei Multithreading dynamischer ausgerichteter Buffer
6       * std::vector<int> v(600);
7       * int* buffer = v.data() + (reinterpret_cast<std::uintptr_t>
8       *                           (v.data()) % 32) / sizeof(int);
9       * oder bei wenigen Threads auf den Stack
10      * __m256i buffer[66]; */
11     int* buff = reinterpret_cast<int *>(buffer);
12     sort_int_sorting_network(arr + left, buff, right - left + 1);
```

```
13      return; }
14    /* avg ist Durchschnitt von größtem und kleinstem Wert im Array */
15    int pivot = choose_avg ? avg : get_pivot(arr, left, right);
16    int smallest = INT32_MAX;       /* kleinster Wert nach Partitionierung */
17    int biggest = INT32_MIN;          /* größter Wert nach Partitionierung */
18    int bound = partition_vectorized_64(arr, left, right + 1, pivot,
19                                                    smallest, biggest);
20    /* Anteil der kleineren Partition am Array */
21    double ratio = (min(right-(bound-1),bound-left)/double(right-left+1));
22    /* falls unbalancierte Teilarrays, Pivot-Berechnung ändern */
23    if (ratio < 0.2) { choose_avg = !choose_avg; }
24    if (pivot != smallest)        /* Werte im linken Teilarray verschieden */
25      qs_core(arr, left, bound - 1, choose_avg, average(smallest, pivot));
26    if (pivot + 1 != biggest)    /* Werte im rechten Teilarray verschieden */
27      qs_core(arr, bound, right, choose_avg, average(biggest, pivot)); }
28
29  inline void quicksort(int *arr, int n) { qs_core(arr, 0, n - 1); }
```

Quelltext A.15: Rekursive Funktion für vektorisierten Quickselect

```
1   inline void qsel_core(int *arr, int left, int right, int k,
2                         bool choose_avg = false, const int avg = 0) {
3     if (right - left < 256) {
4       /* für wenige Elemente C++'s nth_element benutzen */
5       nth_element(arr + left, arr + k, arr + right + 1);
6       return; }
7     /* avg ist Durchschnitt von größtem und kleinstem Wert im Array */
8     int pivot = choose_avg ? avg : get_pivot(arr, left, right);
9     int smallest = INT32_MAX;       /* kleinster Wert nach Partitionierung */
10    int biggest = INT32_MIN;          /* größter Wert nach Partitionierung */
11    int bound = partition_vectorized_64(arr, left, right + 1, pivot,
12                                                    smallest, biggest);
13    /* Anteil der kleineren Partition am Array */
14    double ratio = (min(right-(bound-1),bound-left)/double(right-left+1));
15    /* falls unbalancierte Teilarrays, Pivot-Berechnung ändern */
16    if (ratio < 0.2) { choose_avg = !choose_avg; }
17    if(k < bound){                /* k ist auf der linken Seite von bound */
18      if(pivot != smallest)       /* Werte im linken Teilarray verschieden */
19        qsel_core(arr, left, bound-1, k, choose_avg,average(smallest,pivot));
20    } else {                      /* k ist auf der rechten Seite von bound */
21      if(pivot + 1 != biggest)   /* Werte im rechten Teilarray verschieden */
22        qsel_core(arr, bound, right, k, choose_avg,average(biggest,pivot));}}
23
24  inline void quickselect(int *arr, int n, int k){qsel_core(arr, 0, n-1, k);}
```

Literaturverzeichnis

Baddar, Sherenaz and Kenneth E. Batcher (2011). *Designing sorting networks : a new paradigm*. Springer. ISBN: 1461418518, 9781461418504.

Batcher, Kenneth E. (1968). "Sorting Networks and Their Applications". In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. ACM, pp. 307–314. DOI: 10.1145/1468075.1468121.

Bentley, Jon L. and M. Douglas McIlroy (1993). "Engineering a Sort Function". In: *Software—Practice & Experience* 23(11), pp. 1249–1265. DOI: 10.1002/spe.4380231105.

Blackman, David und Sebastiano Vigna (2018). *Xoroshiro128+*. URL: http://xoshiro.di.unimi.it/xoroshiro128plus.c (besucht am 19. 09. 2018).

Bramas, Berenger (2017). "A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake". In: *International Journal of Advanced Computer Science and Applications* 8(10). DOI: 10.14569/IJACSA.2017.081044.

Bryant, Randal (2016). *Computer systems: A Programmer's Perspective*. 3rd ed. Pearson. ISBN: 013409266X, 9780134092669.

Chhugani, Jatin, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey (2008). "Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture". In: *Proceedings of the VLDB Endowment* 1(2), pp. 1313–1324. DOI: 10.14778/1454159.1454171.

Ciura, Marcin (2001). "Best Increments for the Average Case of Shellsort". In: *Proceedings of the 13th International Symposium on Fundamentals of Computation Theory*. Springer, pp. 106–117. ISBN: 3540424873, 9783540424871.

Codish, Michael, Luís Cruz-Filipe, Thorsten Ehlers, Mike Müller, and Peter Schneider-Kamp (2016). "Sorting networks: To the end and back again". In: *Journal of Computer and System Sciences*. DOI: 10.1016/j.jcss.2016.04.004.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2009). *Introduction to algorithms*. 3rd ed. MIT press. ISBN: 0262033844, 9780262033848.

Daoud, Amjad M., Hussein Abdel-jaber, and Jafar Ababneh (2011). "Efficient Non-Quadratic Quick Sort (NQQuickSort)". In: *Digital Enterprise and Information*

Systems. Ed. by Ezendu Ariwa and Eyas El-Qawasmeh. Springer, pp. 667–675. ISBN: 3642226035, 9783642226038.

Developer Zone (2008). *Intel® Software Developer Zone*. URL: https://software.intel.com/en-us/articles/measuring-instruction-latency-and-throughput (besucht am 27. 07. 2018).

Dietz, Henry Gordon (2010). *The Aggregate Magic Algorithms*. Tech. rep. University of Kentucky. URL: http://aggregate.org/MAGIC/.

Flynn, Michael J. (1972). "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* 21(9), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

Gueron, Shay and Vlad Krasnov (2016). "Fast Quicksort Implementation Using AVX Instructions". In: *The Computer Journal* 59(1), pp. 83–90. DOI: 10.1093/comjnl/bxv063.

Hayes, Timothy, Oscar Palomar, Osman Unsal, Adrian Cristal, and Mateo Valero (2015). "VSR sort: A novel vectorised sorting algorithm & architecture extensions for future microprocessors". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 26–38. DOI: 10.1109/HPCA.2015.7056019.

Herold, Helmut (2017). *Grundlagen der Informatik*. 3. Aufl. Pearson. ISBN: 3868943161, 9783868943160.

Heun, Volker (2003). *Grundlegende Algorithmen: Einführung in den Entwurf und die Analyse effizienter Algorithmen*. 2. Aufl. Vieweg. ISBN: 3528131403, 9783528131401.

Huang, Chun-Yueh, Gwo-Jeng Yu, and Bin-Da Liu (2001). "A hardware design approach for merge-sorting network". In: *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*. Vol. 4, pp. 534–537. DOI: 10.1109/ISCAS.2001.922292.

Inoue, Hiroshi, Takao Moriyama, Hideaki Komatsu, and Toshio Nakatani (2007). "AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors". In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pp. 189–198. DOI: 10.1109/PACT.2007.4336211.

Inoue, Hiroshi and Kenjiro Taura (2015). "SIMD- and Cache-friendly Algorithm for Sorting an Array of Structures". In: *Proceedings of the VLDB Endowment* 8(11), pp. 1274–1285. DOI: 10.14778/2809974.2809988.

Intel Corporation (2016a). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-033.

Intel Corporation (2016b). *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 1. 253665-060US.

Intrinsics Guide (2018). *Intel Intrinsics Guide*. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (besucht am 18. 07. 2018).

Kneusel, Ronald (2018). *Random Numbers and Computers*. 1st ed. Springer. ISBN: 3319776967, 9783319776965.

Knuth, Donald E. (1998). *The Art of Computer Programming: Volume 3: Sorting and Searching*. 2nd ed. Addison-Wesley, Reading. ISBN: 0201896850, 9780201896855.

Lemire, Daniel (2018). *Fast random number generators: Vectorized (SIMD) version of xorshift128+*. URL: https://github.com/lemire/SIMDxorshift (besucht am 20. 09. 2018).

Levin, Stewart A. (1990). "A fully vectorized quicksort". In: *Parallel Computing* 16, pp. 369–373. DOI: 10.1016/0167-8191(90)90074-J.

Li, Peng, David J. Lilja, Weikang Qian, Kia Bazargan, and Marc D. Riedel (2014). "Computation on Stochastic Bit Streams Digital Image Processing Case Studies." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22(3), pp. 449–462.

Nebel, Markus (2012). *Entwurf und Analyse von Algorithmen*. Springer Vieweg. ISBN: 3834819492, 9783834819499.

Nielsen, Frank (2016). *Introduction to HPC with MPI for Data Science*. 1st ed. Springer. ISBN: 3319219022, 9783319219028.

Parberry, Ian (1992). "The Pairwise Sorting Network". In: *Parallel Processing Letters* 2, pp. 205–211. DOI: 10.1142/S0129626492000337.

Parhami, Behrooz (2002). *Introduction to parallel processing: algorithms and architectures*. Kluwer Academic. ISBN: 0306459701, 9780306469640.

C++ Reference (2017). *std::nth_element*. URL: http://cplusplus.com/reference/algorithm/nth_element/ (besucht am 27. 09. 2018).

Ruud, Pas (2017). *Using OpenMP—the next step: affinity, accelerators, tasking, and SIMD*. The MIT Press. ISBN: 0262534789, 9780262534789.

Sedgewick, Robert (2014). *Algorithmen: Algorithmen und Datenstrukturen*. 4. Aufl. Pearson. ISBN: 3868941843, 9783868941845.

Siegel, Howard Jay (1977). "The Universality of Various Types of SIMD Machine Interconnection Networks". In: *SIGARCH Comput. Archit. News* 5(7), pp. 70–79. DOI: 10.1145/633615.810655.

Stone, Harold S. (1978). "Sorting on STAR". In: *IEEE Transactions on Software Engineering* SE-4(2), pp. 138–146. DOI: 10.1109/TSE.1978.231484.

Sullivan, Francis and Jack Dongarra (2000). "Guest Editors' Introduction: The Top 10 Algorithms". In: *Computing in Science & Engineering* 2, pp. 22–23. DOI: 10.1109/MCISE.2000.814652.

Vöcking, Berthold, Helmut Alt, Martin Dietzfelbinger, Rüdiger Reischuk, Christian Scheideler, Heribert Vollmer und Dorothea ER Wagner (2008). *Taschenbuch der Algorithmen*. Springer. ISBN: 3540763937, 9783540763932.

Zagha, Marco and Guy E. Blelloch (1991). "Radix sort for vector multiprocessors". In: *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. ACM, pp. 712–721. DOI: 10.1145/125826.126164.

Zazon-Ivry, Moshe and Michael Codish (2012). "Pairwise Networks are Superior for Selection". In: URL: https://cs.bgu.ac.il/~mcodish/Papers/Sources/pairwiseSelection.pdf.

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Seitens des Verfassers bestehen keine Einwände, die vorliegende Masterarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Jena, den 01.11.2018 _____