Architektur und Programmierung von Grafik- und Koprozessoren

Sortieren auf der GPU

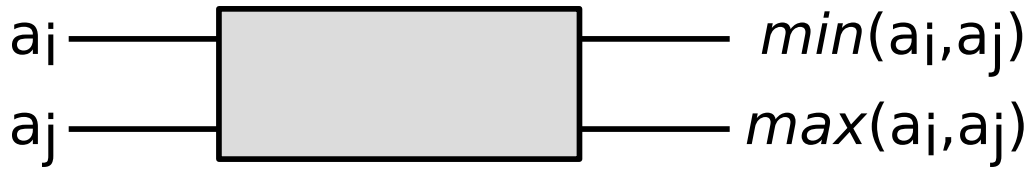
Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

Sortiernetzwerke

Sortiernetzwerke setzen sich aus *Vergleichsmodulen* zusammen:

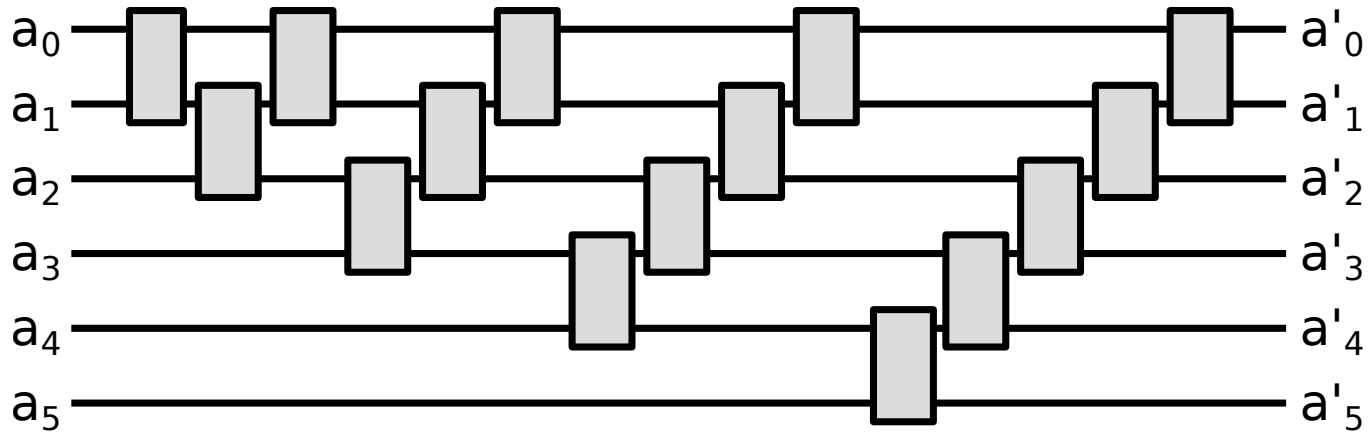


Diese haben zwei Eingabeports (links). Das Vergleichsmodul vergleicht die beiden Eingaben a_i, a_j und vertauscht sie basierend auf der Ordnung. Dies ist eine $O(1)$ Operation. Das Ergebnis liegt auf den Ausgabeports (rechts) an.

Vergleichsmodule sind über *Leitungen* miteinander verbunden.

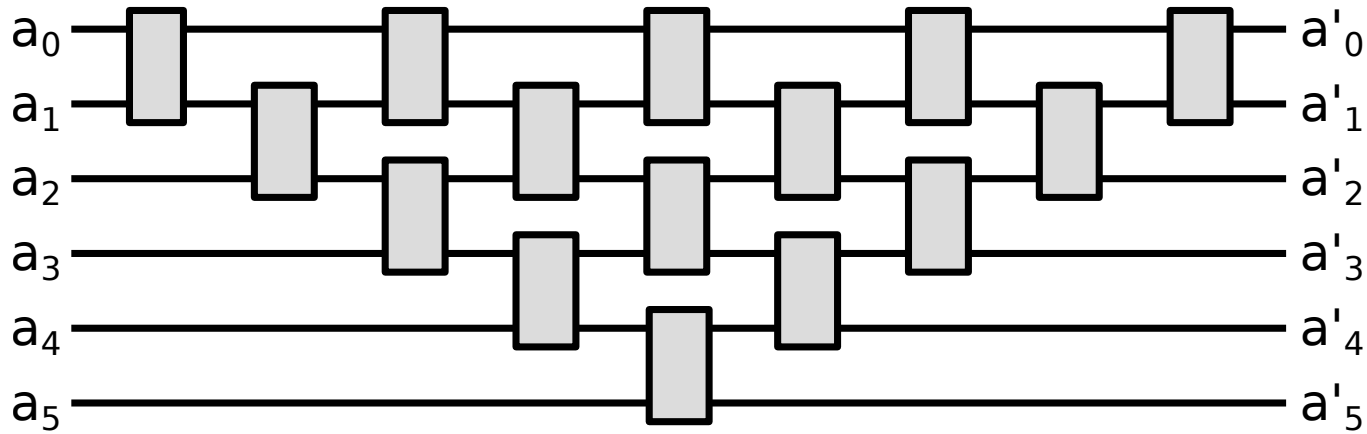
Sortiernetzwerke

Sortiernetzwerk angelehnt an Insertion Sort, das Eingabefolge $A = \{a_0, ..a_5\}$ in sortierte Ausgabefolge $A' = \{a'_0, ..a'_5\}$ überführt.



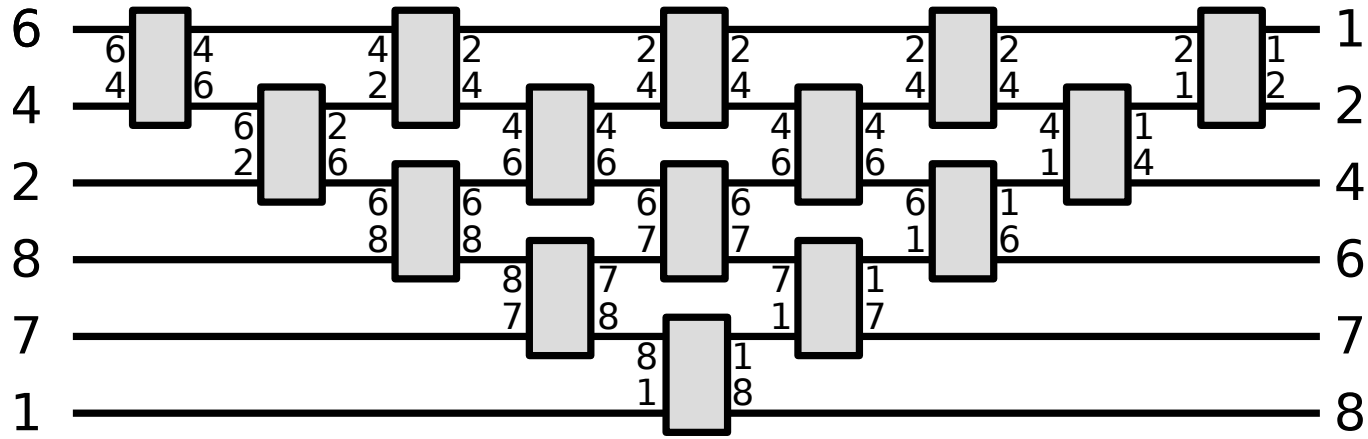
Sortiernetzwerke

Dieses Netzwerk kann man auch etwas anders darstellen, dann sieht man, dass manche Vergleichsmodule parallel ausgeführt werden können.



Sortiernetzwerke

Sortiernetzwerk, das Eingabefolge $A = \{6, 8, 4, 7, 1, 2\}$ in sortierte Ausgabefolge $A' = \{1, 2, 4, 6, 7, 8\}$ überführt.

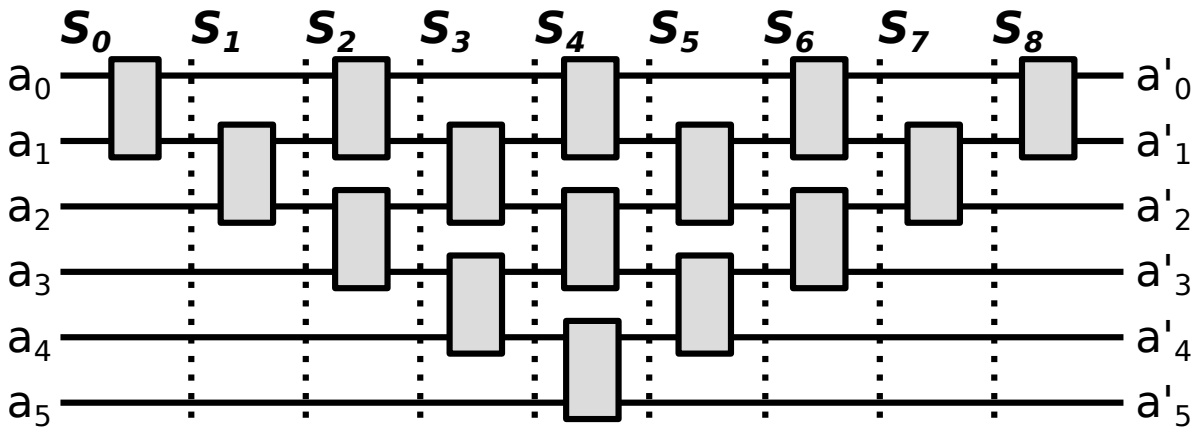


Sortiernetzwerke

Formal:
Vergleichsmodule $C(i, j)$ überführen Eingabepaar $[a_i, a_j]$ in geordnete Paare $[\min(a_i, a_j), \max(a_i, a_j)]$.

Eine *Vergleichsstufe* ist eine Menge *benachbarter Vergleichsmodule* $S_m = \{C(i, j)\}..C(k, l)\}$ sodass für alle i, j, k, l gilt: $i \neq j \neq k \neq l$.
Vergleichsmodule in einer Stufe können parallel ausgeführt werden.

Vergleichsnetzwerk $N_n = S_0, S_1, ..$: Aneinanderreihung von Vergleichsstufen für fixes n.



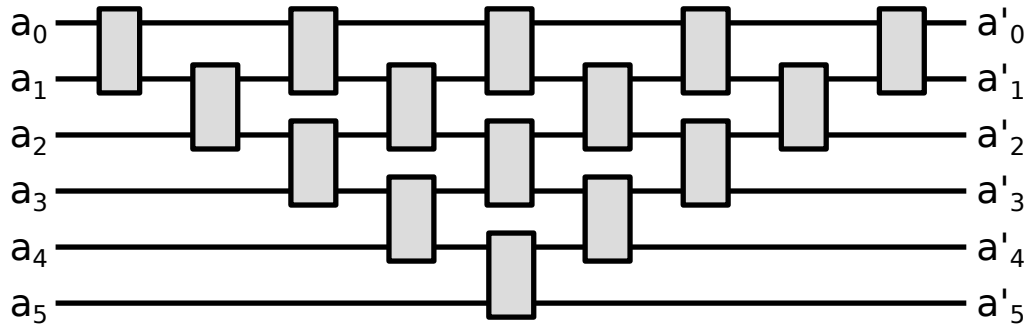
Sortiernetzwerke

Formal:
Wir definieren die *Tiefe* des Vergleichsnetzwerks als dessen Anzahl Vergleichsstufen.

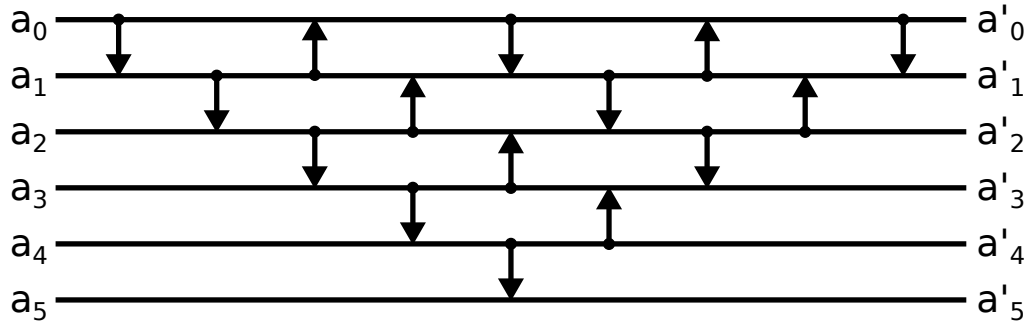
Offensichtlich ist die Zeitkomplexität S des Vergleichsnetzwerks gleich dessen Tiefe.

Sortiernetzwerke

Die Darstellung



impliziert, dass für die a_i, a_j $i < j$ gilt. Dies ist jedoch keine allgemeine Einschränkung, für Vergleiche mit a_i, a_j wobei $i < j$ sowie $i > j$ zulässig, notieren wir das Netzwerk wie folgt:



0-1 Prinzip

Offensichtlich: Sortiernetzwerke sind Vergleichsnetzwerke, die *alle* Eingabesequenzen sortieren.

Es gilt darüber hinaus das **Theorem**:
Ein Vergleichsnetzwerk mit n Eingabeports ist ein Sortiernetzwerk, wenn es alle 2^n möglichen Eingabesequenzen A mit $a_i \in \{0, 1\}$ sortiert (*0-1 Prinzip*).

0-1 Prinzip Beweis

Gegeben sei eine monotone Abbildung $f(x)$, sodass $f(x) \leq f(y)$ falls $x \leq y$. Gegeben sei außerdem ein *Vergleichsnetzwerk* N, das die Folge $A = a_0, ..., a_{n-1}$ in die Folge $A' = a'_0, ..., a'_{n-1}$ überführt.

$\Rightarrow$ N überführt die Folge $f(a_0), ..., f(a_{n-1})$ in die Folge $f(a'_0), ..., f(a'_{n-1})$.

Sei N ein Vergleichsnetzwerk, das 0-1 Folgen korrekt sortiert.

Angenommen, dass für ein beliebiges Paar (a'_i, a'_{i+1}) $(i \in [0..n-1))$ gilt: $a'_i > a'_{i+1}$ (d. h. N sortiert zwar 0-1 Folgen, aber nicht A). Wir wählen nun für f die konkrete Abbildung

$$f(a') = \begin{cases} 0, \text{ falls } a' < a'_i \\ 1, \text{ falls } a' \geq a'_i \end{cases}$$

für ein beliebiges $a'_i \in A'$. Dann wäre $f(a'_0), ..., f(a'_{n-1})$ eine nicht sortierte 0-1 Folge ($\Rightarrow$ Widerspruch). $\square$

Bitonic Sort

Sei $A = a_0, ...a_{n-1}$ mit $a_i \in \{0, 1\}$. Die *0-1 Folge A* heißt *bitonisch*, wenn für

$$A = a_0, ..., a_l, a_{l+1}, ..., a_{n-1}$$

gilt, dass $a_0, ..., a_l$ monoton steigend und $a_{l+1}, ..., a_{n-1}$ fallend, oder (umgekehrt), dass $a_0, ..., a_l$ monoton fallend und $a_{l+1}, ..., a_{n-1}$ steigend.

Die 0-1 Folgen

$$A_1 = 0, 0, 0, 1, 1, 0, 0$$
$$A_2 = 1, 0, 1$$
$$A_3 = 0, 0, 0, 0$$
$$A_4 = 1, 1$$
$$A_5 = 0, 1$$

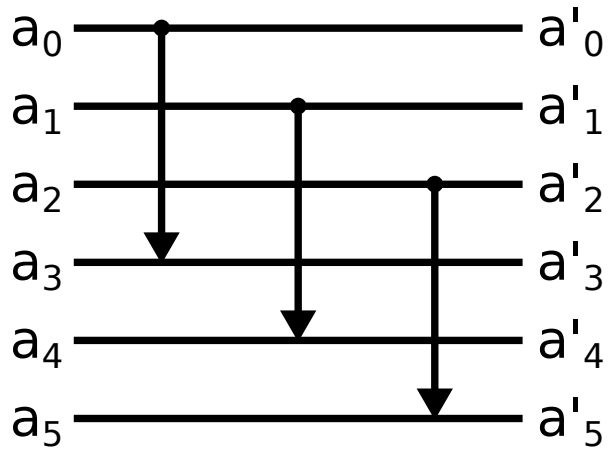
erfüllen etwa diese Eigenschaft.

Bitonic Sort

Wir betrachten das Vergleichsnetzwerk

$$N_n = C\left(0, \frac{n}{2}\right), C\left(1, \frac{n}{2}+1\right), ..., C\left(\frac{n}{2}-1, n-1\right) \qquad (1)$$

für $n \in \mathbb{N}$ gerade, z. B. N_6:



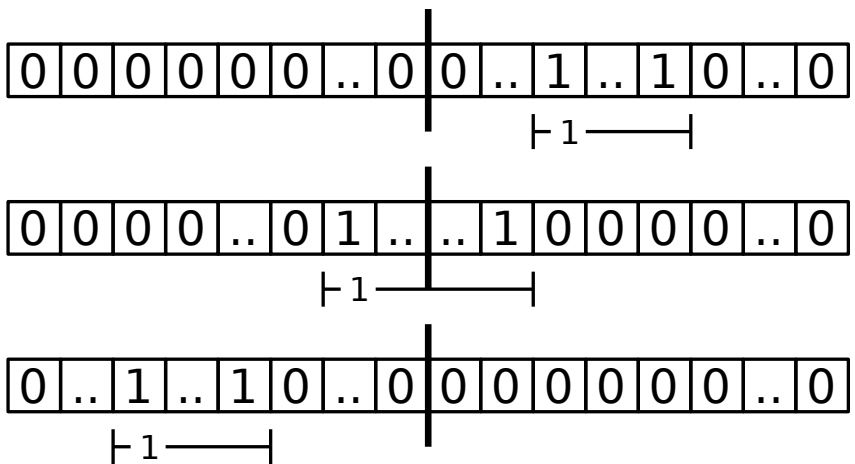
Bitonic Sort

Theorem:
Sei $a(n) = a_0, a_1, ..., a_{n-1}$ eine bitonische 0-1 Folge mit n gerade. Wendet man das Vergleichsnetzwerk N_n auf $a(n)$ an, erhält man

$$b(n) = a'_0, ..., a'_{\frac{n}{2}-1}, a''_0, ..., a''_{\frac{n}{2}-1},$$

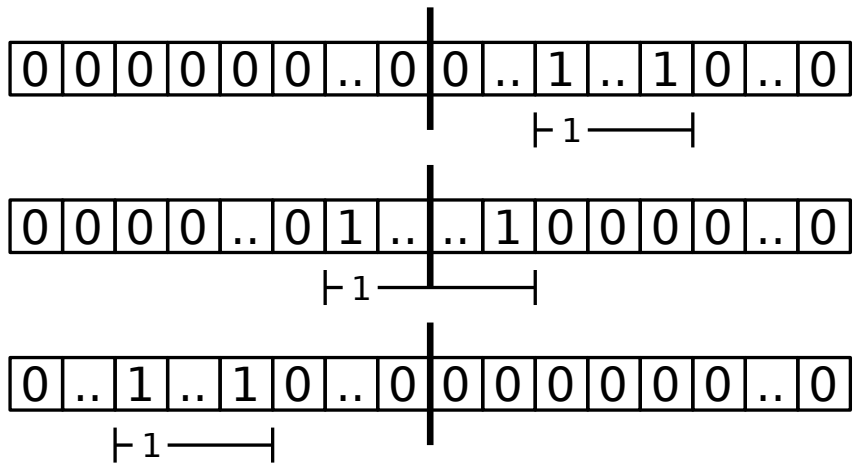
sodass $a'_i \leq a''_i \ \forall \ i = 0, 1, ..., \frac{n}{2} - 1$. Die Teilfolgen $a'_0, ..., a'_{\frac{n}{2}-1}$ und $a''_0, ..., a''_{\frac{n}{2}-1}$ sind ebenfalls bitonisch.

Beweis:
Mühsam. Nicht-triviale Folgen $a(n)$ haben eine der drei Formen (oder deren negierte Varianten):



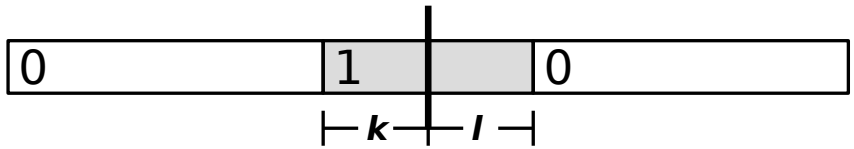
Bitonic Sort

Die negierten und trivialen Varianten inbegriffen müssen wir das Theorem also für acht Arten von Folgen beweisen. Wir zeigen es für die Variante in der Mitte, bei der der 1er Block linksseitig und rechtsseitig über die Position $\frac{n}{2}$ herausragt. Die übrigen Varianten (ähnlich) sind Übungsaufgabe.



Bitonic Sort

Wir betrachten also Folgen der Form



sodass gilt

$$a_0, \ldots, a_{\frac{n}{2}-k-1} = 0$$

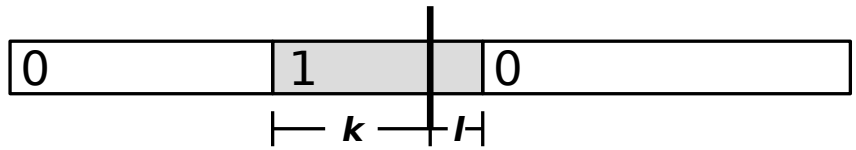
$$a_{\frac{n}{2}-k}, \ldots, a_{\frac{n}{2}+l-1} = 1$$

$$a_{\frac{n}{2}+l}, \ldots, a_{n-1} = 0$$

Bitonic Sort

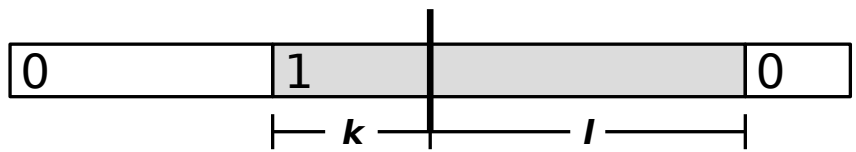
wir müssen zwei weitere Fälle unterscheiden:

$$k + l \leq \frac{n}{2}$$



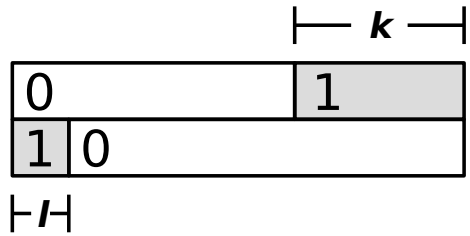
und

$$k + l > \frac{n}{2}$$

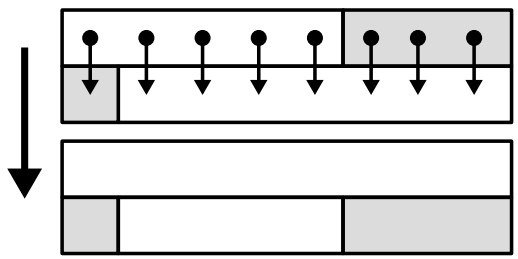


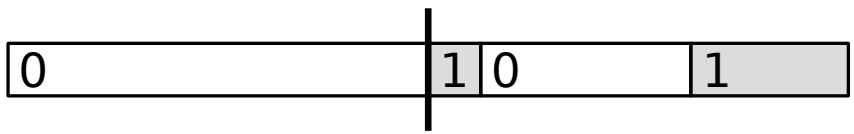
Bitonic Sort

Wir zeichnen für $k + l \leq \frac{n}{2}$ die beiden Teilfolgen $a_0, ..., a_{\frac{n}{2}-1}$ und $a_{\frac{n}{2}}, a_{n-1}$ untereinander:

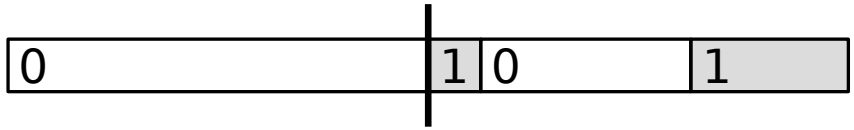


Wir wenden nun N_n auf die Folge $a(n)$ an. Man sieht leicht, dass sich zwei neue Teilfolgen ergeben, die das Theorem erfüllen.





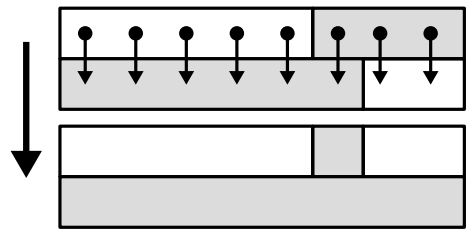
Bitonic Sort



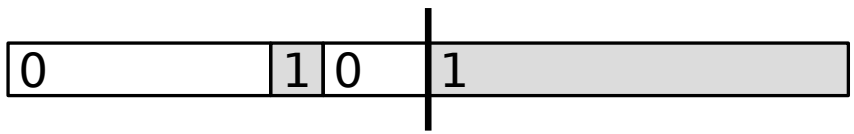
Alle Elemente der linken Teilfolge sind $\leq$ den korrespondierenden Elementen aus der rechten Teilfolge. Beide Teilfolgen sind bitonisch.

Bitonic Sort

Wir können ganz ähnlich für den Fall $k + l > \frac{n}{2}$ verfahren und erhalten die Skizze:



Auch hier sind alle Elemente der linken Teilfolge $\leq$ den korrespondierenden Elementen aus der rechten Teilfolge und beide Teilfolgen sind bitonisch:



Wir können sehr ähnlich die verbleibenden Varianten zeigen, woraus die Behauptung folgt. □

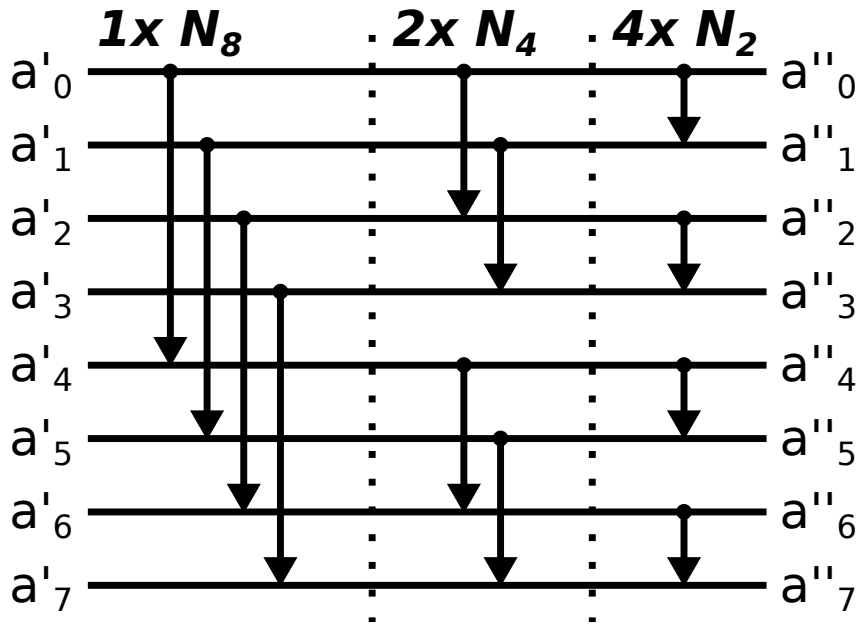
Bitonic Sort

Bemerkung

- Das Vergleichsnetzwerk N_n partioniert also zwei 0-1 Folgen so, dass alle Elemente in der linken Hälfte $\leq$ den entsprechenden Elementen in der rechten Hälfte sind.
- Die beiden neu erzeugten Folgen haben die gleichen Monotonieeigenschaften wie die Ausgangsfolge.
- Wegen des 0-1 Prinzips gelten die Eigenschaften auch für beliebige bitonische Folgen.
- Offensichtlich kann man mit Hilfe des N_n Vergleichsnetzwerks ein Divide & Conquer Verfahren (und ein zugehöriges Sortiernetzwerk) formulieren, das jede *bitonische* Eingabesequenz sortiert.

Bitonic Sort

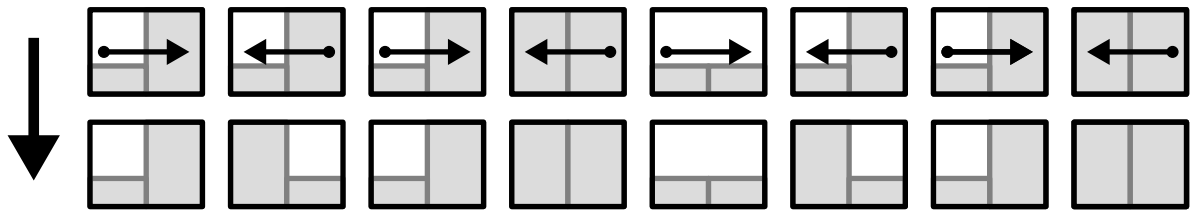
Durch rekursives Ausführen von N_n kann die 0-1 Folge $a'(n),\ n = 2^k$ sortiert werden, so diese *bitonisch* ist. Exemplarisch für $n = 8$:



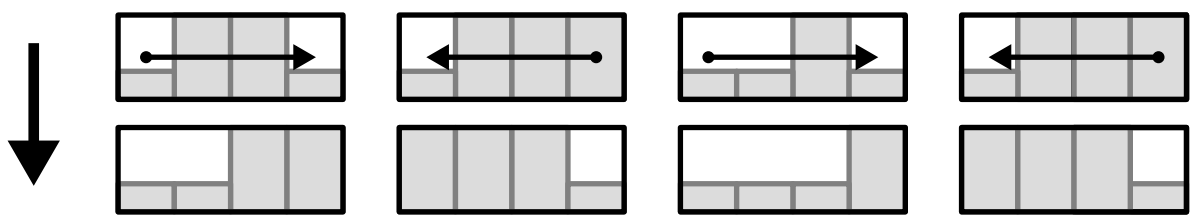
Um eine beliebige 0-1 Folge $a(n)$ der Länge $n = 2^k$ zu sortieren, müssen wir sie also erst umformen, sodass sie, wie $a'(n)$, bitonisch ist.

Bitonic Sort

Wir stellen zunächst fest, dass jede 0-1 Folge $a(n)$ der Länge 2 bitonisch ist. Wir teilen daher die Eingabesequenz in k Paare auf und sortieren diese abwechselnd aufsteigend und absteigend mit dem N_2 Vergleichsnetzwerk:

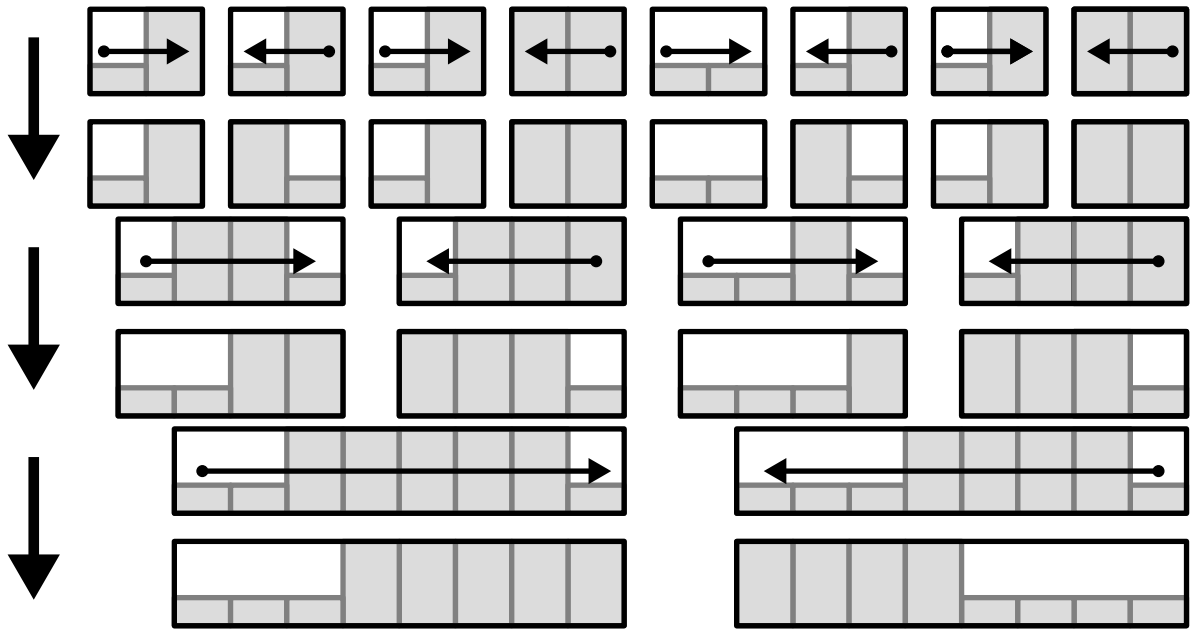


Wir *mergen* die Paare zu bitonischen Folgen der Länge 4. Diese sortieren wir wieder abwechselnd aufsteigend und absteigend. Wir applizieren dazu das rekursive Vergleichsnetzwerk von vorhin:



Bitonic Sort

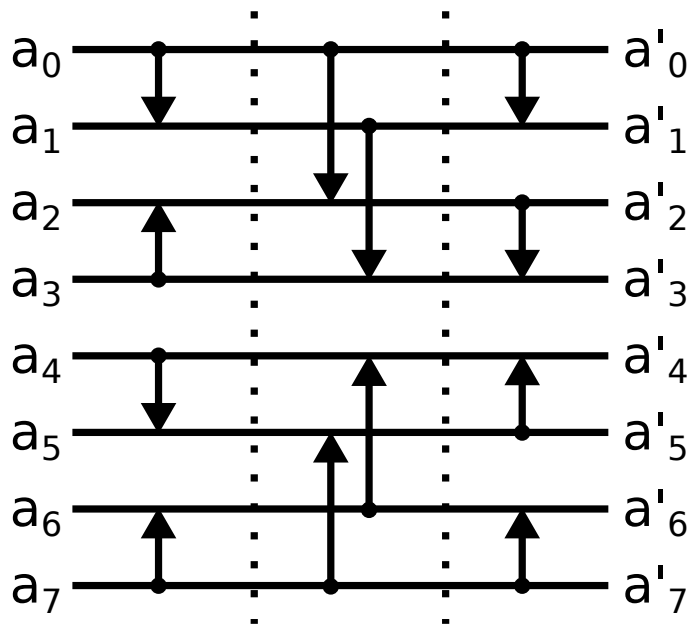
Dies wiederholen wir so oft, bis wir zwei sortierte Teilfolgen der Länge $\frac{n}{2}$ haben, die erste aufsteigend sortiert, die zweite absteigend sortiert.



Die sich ergebende Folge $a'(n)$ mit Länge n ist bitonisch und lässt sich mit dem rekursiven Vergleichsnetzwerk von vorhin sortieren.

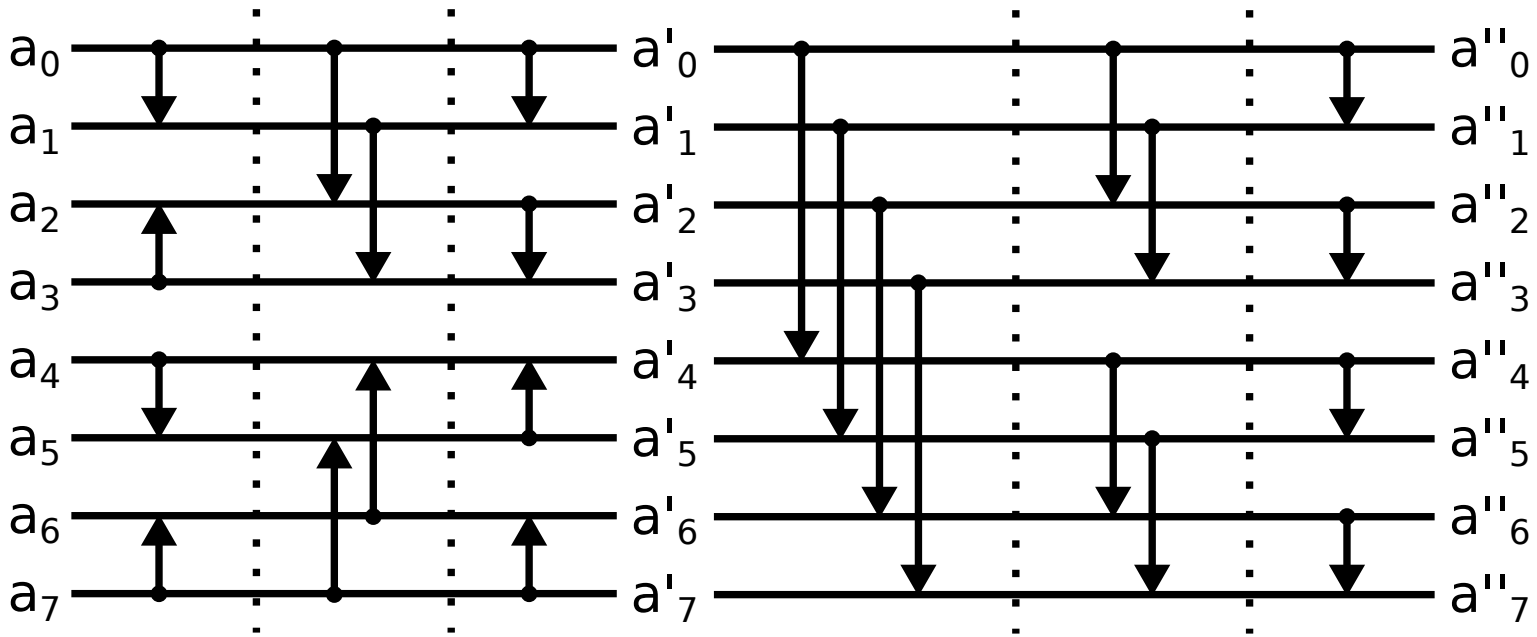
Bitonic Sort

Die Abbildung zeigt das Vergleichsnetzwerk, das jede unsortierte Folge $a(n)$ mit Länge 8 in eine bitonische Folge $a'(n)$ überführt.



Bitonic Sort

Das zusammengesetzte Vergleichsnetzwerk überführt die 0-1 Folge zunächst in eine bitonische Folge und sortiert diese dann. Wegen des 0-1 Prinzips ist das Vergleichsnetzwerk ein Sortiernetzwerk für beliebige Eingabesequenzen mit Länge $n = 2^k$.



Bitonic Sort

Komplexität

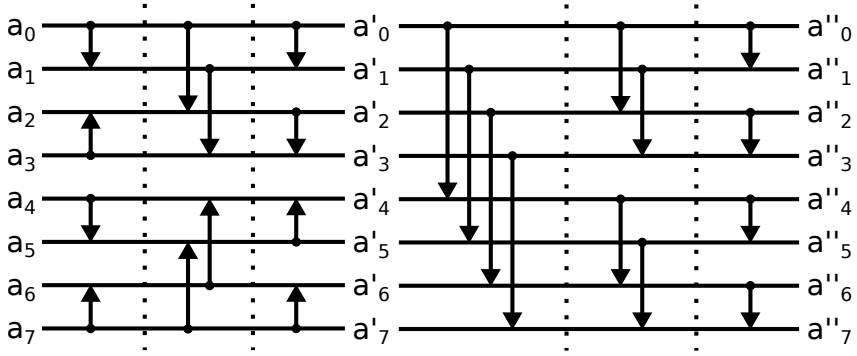
Anzahl *Vergleichsstufen*: $\frac{1}{2}\log n \times (\log n + 1)$.

Anzahl *Vergleichsmodule* pro Vergleichsstufe: $\frac{n}{2}$.

Daher:

- Arbeitskomplexität: $W(n) = O(n\log^2 n)$.
- Zeitkomplexität: $S(n) = O(\log^2 n)$.

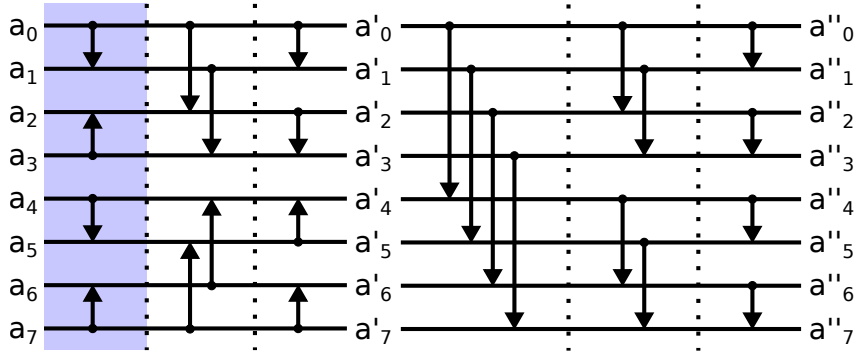
Bitonic Sort Beispiel



Sortiere die Folge

2	3	4	5	6	7	8	9

Bitonic Sort Beispiel



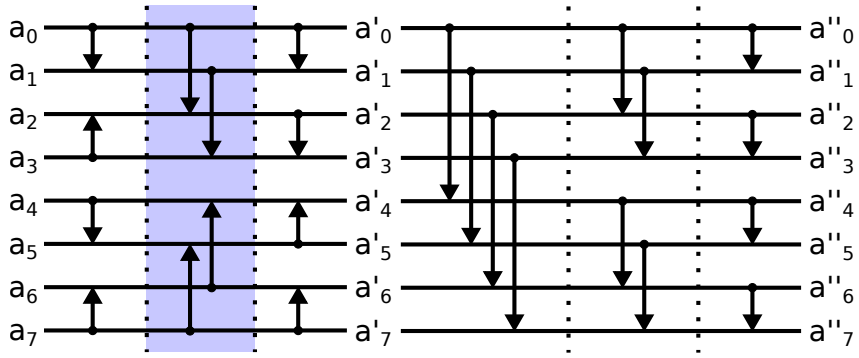
vor Stufe 1:

2	3	4	5	6	7	8	9

nach Stufe 1:

2	3	5	4	6	7	9	8

Bitonic Sort Beispiel



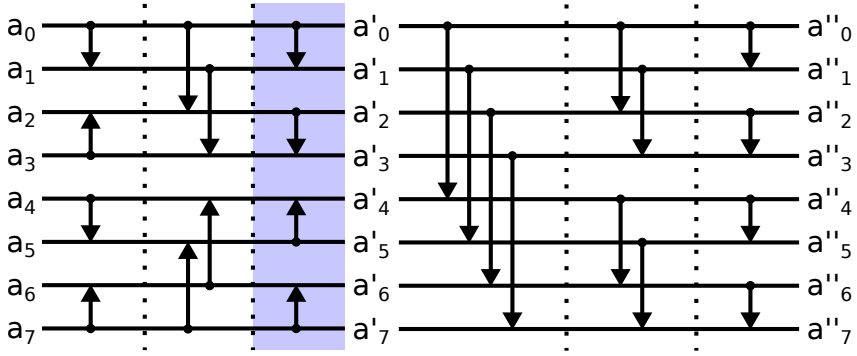
vor Stufe 2:

2	3	5	4	6	7	9	8

nach Stufe 2:

2	3	5	4	9	8	6	7

Bitonic Sort Beispiel



vor Stufe 3:

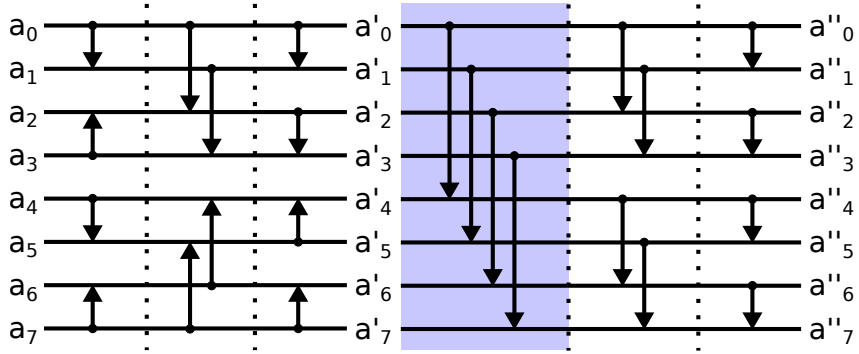
2	3	5	4	9	8	6	7

nach Stufe 3:

2	3	4	5	9	8	7	6

Die Folge ist nun bitonisch.

Bitonic Sort Beispiel



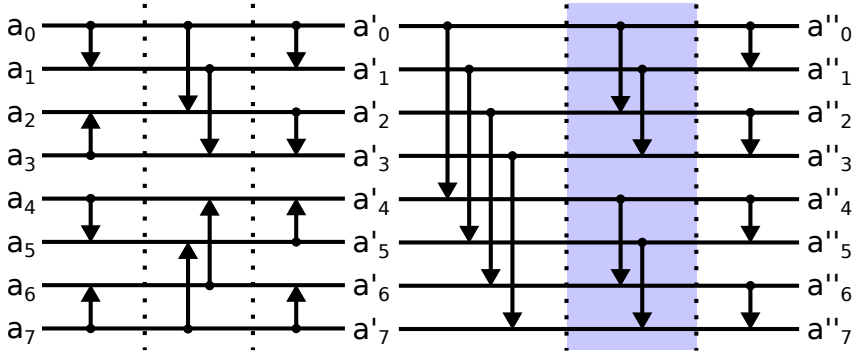
vor Stufe 4:

2	3	4	5	9	8	7	6

nach Stufe 4:

2	3	4	5	9	8	7	6

Bitonic Sort Beispiel



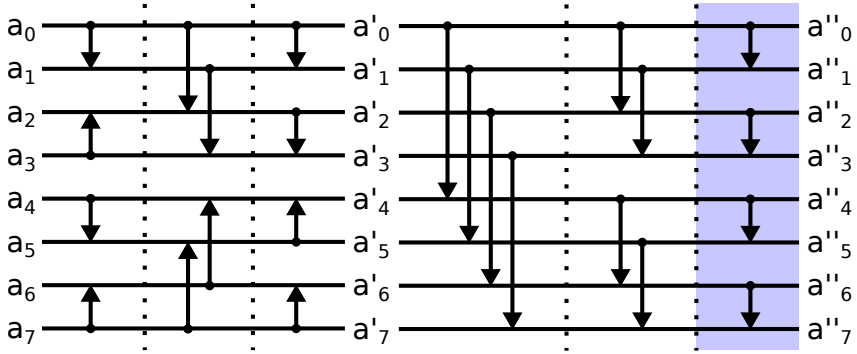
vor Stufe 5:

2	3	4	5	9	8	7	6

nach Stufe 5:

2	3	4	5	7	6	9	8

Bitonic Sort Beispiel



vor Stufe 6:

2	3	4	5	7	6	9	8

nach Stufe 6:

2	3	4	5	6	7	8	9

Die Folge ist nun sortiert.

Bitonic Sort Implementierung mit CUDA

Vorgehen

Fallstudie: Bitonic Sort mit CUDA

Entwickle erst *korrekte* Version des Algorithmus in C++ für CPU.
Vermeide folgende Sprachkonstrukte:

- Rekursion.
- Dynamische Speicherallokation.

Optimiere CPU Version so weit, dass sie sich mit OpenMP einfach parallelisieren lässt. Portiere *dann erst* nach CUDA.

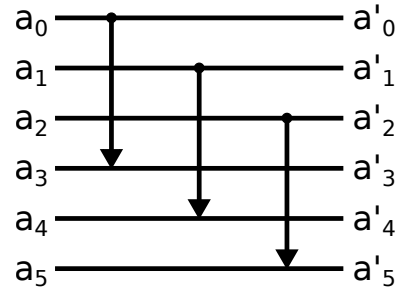
Erste Version

(1/2)

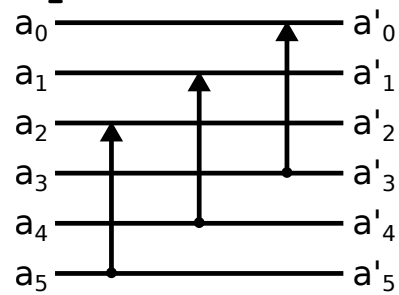
N_n Funktion mit Parameter für Ausführungsrichtung.

```
enum Direction { Up = 0, Down = 1 };

template <typename It>
void N(It first, It last, Direction dir) {

    auto n = std::distance(first, last);

    if (dir == Down)
        for (std::size_t i = 0; i < n / 2; ++i)
            if (first[i + n / 2] < first[i])
                swap(first[i], first[i + n / 2]);
    else
        for (std::size_t i = 0; i < n / 2; ++i)
            if (first[i] < first[i + n / 2])
                swap(first[i], first[i + n / 2]);
}
```

Down:



Up:



Erste Version

(2/2)

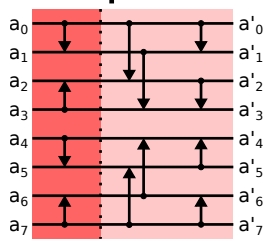
```
template <typename It>
void bitonic_sort_cpu_00(It first, It last) {
    auto n = std::distance(first, last);

    // Make bitonic sequence
    for (std::size_t i = 2; i < n; i *= 2) {
        Direction dir = Down;
        for (std::size_t j = i; j >= 2; j /= 2)
        {
            for (std::size_t k = 0; k < n; k += j)
            {
                N(first + k, first + k + j, dir);
                if ((k + j) % i == 0)
                    dir = Direction(~dir);
            }
        }
    }

    // Sort bitonic sequence
    for (std::size_t j = n; j >= 2; j /= 2)
        for (std::size_t k = 0; k < n; k += j)
            N(first + k, first + k + j, Down);
}
```

Erste Version

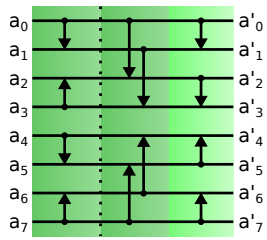
(2/2)

```
template <typename It>
void bitonic_sort_cpu_00(It first, It last) {
    auto n = std::distance(first, last);

    // Make bitonic sequence
    for (std::size_t i = 2; i < n; i *= 2) {
        Direction dir = Down;
        for (std::size_t j = i; j >= 2; j /= 2)
        {
            for (std::size_t k = 0; k < n; k += j)
            {
                N(first + k, first + k + j, dir);
                if ((k + j) % i == 0)
                    dir = Direction(~dir);
            }
        }
    }

    // Sort bitonic sequence
    for (std::size_t j = n; j >= 2; j /= 2)
        for (std::size_t k = 0; k < n; k += j)
            N(first + k, first + k + j, Down);
}
```

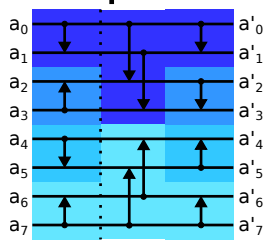
Loop over `i`:



Loop over `j`:



Loop over `k`:



Optimierung 1

Entferne Datenabhängig zu `dir`:

```
// Make bitonic sequence
for (std::size_t i = 2; i < n; i *= 2) {
  Direction dir = Down;
  for (std::size_t j = i; j >= 2; j /= 2)
  {
    for (std::size_t k = 0; k < n; k += j)
    {
      N(first + k, first + k + j, dir);
      if ((k + j) % i == 0)
        dir = Direction(~dir);
    }
  }
}
```

```
// Make bitonic sequence
for (std::size_t i = 2; i < n; i *= 2) {
  for (std::size_t j = i; j >= 2; j /= 2)
  {
    for (std::size_t k = 0; k < n; k += j)
    {
      if (((k ^ j) & i) == 0)
        N(first + k, first + k + j, Down);
      else
        N(first + k, first + k + j, Up);
    }
  }
}
```

Optimierung 2

Merge “`make_sequence()`” und “`sort_sequence()`”, inline Funktion `N()`:

```
template <typename It>
void bitonic_sort_cpu_02(It first, It last) {
    auto n = std::distance(first, last);

    for (std::size_t i = 2; i <= n; i *= 2) // <= n (includes merge)
    {
        for (std::size_t j = i; j >= 2; j /= 2)
        {
            for (std::size_t k = 0; k < n; k += j)
            {
                for (std::size_t l = k; l < k + j / 2; ++l)
                    if (i == n || ((k ^ j) & i) == 0)
                        if (first[l + j / 2] < first[l])
                            swap(first[l], first[l + j / 2]);
                    else
                        if (first[l] < first[l + j / 2])
                            swap(first[l], first[l + j / 2]);
                    }
                }
            }
        }
    }
}
```

Optimierung 3

Führe Schleifen über `k` und `l` zusammen $\Rightarrow$ alle Vergleichsoperationen auf einer Vergleichsstufe parallelisierbar.

```
template <typename It>
void bitonic_sort_cpu(It first, It last) {
    auto n = std::distance(first, last);

    for (std::size_t i = 2; i <= n; i *= 2)
    {
        for (std::size_t j = i / 2; j >= 1; j /= 2)
        {
            #pragma omp parallel for
            for (std::size_t k = 0; k < n; ++k)
            {
                if ((k ^ j) > k)
                {
                    if ((k & i) == 0)
                        if (first[k ^ j] < first[k])
                            swap(first[k], first[k ^ j]);
                    else
                        if (first[k] < first[k ^ j])
                            swap(first[k], first[k ^ j]);
                    }
                }
            }
        }
    }
}
```

Bitonic Sort CPU Version

Vergleiche mit `std::sort`, sortiere 2^{20} Integers.

Erste Version mit Datenabhängigkeit: `std::sort` 10× schneller.

Zweite Version ohne Datenabhängigkeit: `std::sort` 3× schneller.

Dritte Version, ein Thread: `std::sort` 4× schneller.

Dritte Version, acht Threads: `std::sort` 1.3× langsamer.

Bemerkungen:

- Zeitaufwand für Optimierung 2 Stunden ⇒ ganz ok, `std::sort` ist hochoptimiert.
- Die parallelisierte Version *skaliert*. Jetzt erst macht es Sinn, Code auf GPU zu portieren.

CUDA Portierung

Innere Schleife über `k` in Kernel, äußere Schleifen auf CPU.

```
template <typename It>
__global__
void bitonic_sort_kernel(It first, It last, std::size_t i, std::size_t j)
{
    unsigned k = blockIdx.x * blockDim.x + threadIdx.x;

    if ((k ^ j) > k) {
        if ((k & i) == 0)
            if (first[k ^ j] < first[k])
                swap(first[k], first[k ^ j]);
        }
        else
            if (first[k] < first[k ^ j])
                swap(first[k], first[k ^ j]);
    }
}

template <typename It>
void bitonic_sort_cuda(It first, It last) {
    auto n = std::distance(first, last);
    for (std::size_t i = 2; i <= n; i *= 2) {
        for (std::size_t j = i / 2; j >= 1; j /= 2) {
            unsigned threads_per_block = min(n, 1024);
            unsigned blocks = div_up(n, threads_per_block);

            bitonic_sort_kernel<<<blocks, threads_per_block>>>(first, last, i, j);
        }
    }
}
```

Bitonic Sort CUDA Version

- Diese recht unoptimierte Version von Bitonic Sort ist “out of the box” 15× schneller als `std::sort`.
- Es gibt noch eine Menge Optimierungspotential. Anregungen:
  - Shared Memory.
  - Dynamic Parallelism.
- Fallstudie soll Vorgehen illustrieren, wie man Code auf GPU portiert.
  - Wenn man wirklich auf der GPU sortieren will: `thrust::sort()`.

Recap

- Paralleles Sortieren auf GPUs exemplarisch mit Bitonic Sort.
- Sortiernetzwerk $\Rightarrow$ die Vergleiche, die Bitonic Sort durchführt, stehen a priori fest.
  - Daher u. a. auch gut geeignet für Integrierte Schaltungen.
- Sortieren auf GPUs als Fallstudie. Schnelle Sortieralgorithmen in Bibliotheken wie `thrust`.

Literaturempfehlungen

- Donald E. Knuth: The Art of Computer Programming: Sorting and Searching (Vol. 3) 2nd ed. (1973)