FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Development of parallel sorting algorithms for GPU
Student:	Xuan Thang Nguyen
Supervisor:	Ing. Tomáš Oberhuber, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2021/2022

Instructions

1. Study the basics of programming GPU using CUDA.
2. Learn the fundamentals of the development of parallel algorithms with TNL library (www.tnl-project.org).
3. Learn and understand parallel sorting algorithms, namely bitonic sort and quick sort.
4. Implement both algorithms into TNL library to run on CPU and GPU.
5. Implement unit tests for testing correctness of the implemented algorithms.
6. Perform measurement of speed-up compared to sorting algorithms in the STL library and GPU implementation [1].

[1] https://github.com/davors/gpu-sorting

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 30 November 2020 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Development of parallel sorting algorithms for GPU

Nguyen Xuan Thang

Department of Theoretical Computer Science
Supervisor: Ing. Tomáš Oberhuber, Ph.D.

May 13, 2021

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Oberhuber, Ph.D. for his support, guidance and advices throughout the whole time of creating this thesis.

My gratitude also goes to my family that helped me during these hard times.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 13, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Nguyen, Xuan Thang. *Development of parallel sorting algorithms for GPU.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce se zabývá vybranými paralelními řadícími algoritmy vhodnými pro implementaci na GPU. Jedná se konkrétně o Bitonic sort a Quicksort. Bitonic sort, i když má vyšší časovou složitost, je vhodným kandidátem na řazení malých posloupností. Paralelní Quicksort je rychlejší pro větší vstupy, ale potřebuje $\Theta(n)$ paměti na pomocné pole pro přeskládání. Oba algoritmy jsou popsány a implementovány pro GPU od NVIDIA s pomocí CUDA API a TNL knihovny. Jako jazyk byl zvolen C++. U Bitonic sortu je navíc představena varianta, která využívá jen lambda funkce a odprošťuje se tak od kontejneru dat. Všechny implementace jsou řádně otestovány, změřeny a porovnány s jinými implementacemi, které jsou dostupné pro CPU a GPU.

Klíčová slova paralelní řazení, GPU, CUDA, C++, TNL, Bitonic sort, Quicksort

Abstract

This thesis is about selected sorting algorithms suitable for GPU implementation. The chosen algorithms are Bitonic sort and Quicksort. Although Bitonic sort has a worse theoretical time complexity, it is a suitable candidate for sorting smaller inputs. Quicksort is faster for bigger inputs, but for parallel implementation, $\Theta(n)$ auxiliary memory is needed because it is an out-of-place algorithm. Both algorithms were studied and then implemented in C++ extended with CUDA API with the help of TNL library. For Bitonic sort, a version that only uses lambda functions is introduced. The resulting work was then tested, measured, and compared with other CPU and GPU implementations.

Keywords parallel sort, GPU, CUDA, C++, TNL, Bitonic sort, Quicksort

Summary

Motivation

Sorting is an important operation used in many algorithms, but a single-threaded implementation can take a long time to process big inputs. For this reason, GPUs can be used to sort a sequence in parallel and gain speed-up.

Goals

The goal of this thesis is to implement a parallel version of selected sorting algorithms, namely Bitonic sort and Quicksort. These two functions will be running on GPU using CUDA with the help of TNL library [1]. The implementations are to be tested, measured, and compared against other known CPU and GPU implementations.

Method

To explain the algorithms, GPU architecture, TNL library, and CUDA are first explained. Then, the theory around sorting is introduced and afterward, the algorithms themselves.

The first part of the implementation describes parts of Bitonic sort and shows where speed-up was gained with the use of faster shared memory. In the next part, Quicksort kernels are explained in detail and all steps necessary to partition a task are described. The use of shared memory and its methods of gaining speed-up is also explained.

Results

The resulting work contains an efficient version of Bitonic sort that can be called both from CPU and GPU. Measurements show that this version's Bitonic sort can rival the implementation provided by CUDA SDK [2]. For big inputs, Bitonic quickly loses against other algorithms with better time complexity. Our parallel Quicksort runs faster than the original solution implemented by Cederman et al. [3] but loses against Manca et al.'s [4] optimized implementation of Quicksort. TNL Quicksort was also compared against `thrust::sort`[5] available in the CUDA toolkit. The results show worse performance than the highly optimized `thrust::sort`, but this stems from the fact that the function from `thrust` implements Radix sort internally.

Conclusion

Both Bitonic sort and Quicksort were properly tested and measured and are ready to be added into the TNL library. Although the results do not show the best performance, the speed-up compared to CPU sort is still noticable.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Sorting problem is one of the most studied fields in computer science with a wide variety of applications. All libraries that deal with computation and numbers usually contain a sorting function to be used. Sorting also can be applied to many problems, either to simplify the problem or when the sorted property is required. If a search operation is used often, it might be better to give order to the input and use binary search to find elements faster.

As shown, sorting is very useful, and it is meaningful to study this problem. In some cases, fast sorting can have real-time impact. For this reason, this thesis will present implementations of sorting functions that can run faster than single threaded sorts.

A sorting algorithm that only uses comparisons has to make $\Omega(n \log n)$ compares and as such will run in $\Omega(n \log n)$ time. However, this limitation only applies to single-threaded implementations. With the help of more compute units, it is possible to occupy the hardware and spread out the load to gain speed-up. A CPU, usually labeled as a multi-core device, has only tens of cores to use at the same time and can only handle tens to low hundreds of threads at best. GPUs on the other hand can handle thousands of threads and are generally more suitable for mass parallelization.

The thesis' main goal is to implement selected sorting algorithms suitable for GPU and create a solution ready to be incorporated into the TNL library [1]. The implementation, as part of a templated library, has to be able to sort data of arbitrary type and not only numeric values. The code has to be implemented in C++ with the help of CUDA as this platform is used in the library.

In the first part of the theoretical section, the necessary hardware and software will be introduced. Then the sorting problem itself is explained and improvements are proposed. A general overview of existing GPU sorting implementations will also be present. Not all algorithms are suitable to be parallelized, for this reason, the advantages and disadvantages of the chosen algorithms are explained.

For the algorithms themselves (Bitonic sort and Quicksort), a whole chapter is dedicated to their theory and how the algorithm will run. Bitonic sort, albeit being intensively studied by specialists, is not very known by the outside community. Because of this, the algorithm is described in great detail and a proof is given. Quicksort, while being well-known, has a different implementation compared to its CPU version and the parallelization part is not as straightforward compared to Bitonic sort. For this reason, this thesis will also describe in detail the steps necessary to partition an input and how to use the available resources efficiently.

In the implementation part, the code itself will be studied and the main choices made explained. The algorithms are then further enhanced to utilize faster shared memory available to threads during run-time and then problematic spots will be analyzed.

Finally, this thesis will present experimental results measured on a professional grade GPU and then, comparison between the implementation and other CPU and GPU sorts will be shown.

Chapter 2

Preliminaries

2.1 GPU architecture

A GPU (Graphics processing unit) is a specialized device that works with CPU (Central processing unit) to compute tasks. The main difference between them is how each device is designed. CPUs are highly optimized to run at high frequency using only one or a few threads on one processing unit. A thread is a sequence of instructions that need to be carried out and a processing unit is the hardware that can perform the operations. A CPU, to achieve high throughput, uses many tricks and optimizations such as data and instruction caching, out-of-order execution or branch predicting. A GPU on the other hand relies on simpler hardware but contains more computing units. A CPU by design devotes most of the surface to data cache and other transistors to optimize flow control and minimize data access latency while a GPU occupies the space available with hardware necessary to do calculations. This allows GPU devices to start thousands of threads to use the high number of computing units to process data in parallel and mask data latency this way [6].

2.2 CUDA

CUDA is a computing platform and programming model that allows the programmer to utilize NVIDIA GPUs [6]. The CUDA toolkit not only contains a software environment extending C++ but also other tools to help the programmer develop efficient programs on GPUs, such as debugging and profiling applications. With CUDA, it is also necessary to install the correct driver to enable communication with the GPU. The CUDA API is extensive and is in detail described in the CUDA guide documentation [6]. In this thesis, only the important parts of the CUDA API will be described to give a brief overview.

2.2.1 Thread grouping

The fundamental building blocks of a CUDA program are kernels. Kernels are CUDA C++ extended functions that are executed in parallel by n different CUDA threads [6]. Threads that execute a kernel are grouped into a *block* and the size of the block is set by the programmer during kernel launch. The size of the block is usually set as a multiplier of 32 because of warp execution. A warp is a group of 32 threads within a block that have consecutive thread *id* (e.g. thread 0...31 create a warp).

Each thread in a kernel has access to multiple hidden variables available from anywhere in the device code to identify themselves. The variable `blockDim` holds information about how many threads there are in a block and `threadIdx` is used to identify the thread in a block. To check how many blocks were launched, all threads have access to `gridDim`. It should be noted however, that all three previously mentioned constants are 3 dimensional structures with three fields — x, y, z. For simplicity, the implementation will only use the `x` field to identify a thread inside a CUDA block.

There are also limitations on how many threads can be launched. Each CUDA block has a limited amount of shared memory and registers given by the hardware and the resources need to be divided between threads. For this reason, on the current GPUs, the number of threads per block is limited to 1024 [6].

2.2.2 Memory hierarchy

When accessing memory while programming with CUDA, it is necessary to differentiate between GPU memory and CPU memory. CPU memory can be allocated using `malloc` or `new`, but to allocate memory on GPU, CUDA API has to be used. An allocating operation returns a pointer to the first available memory address. A problem arises when pointers to GPU and CPU memory are mixed and not differentiated. Memory in GPU lies in a different address space than CPU memory, as such, accessing a GPU memory from CPU can lead to segmentation fault. The same thing happens when a CUDA thread tries to access CPU memory. To access a GPU variable from CPU, first, the block of memory needs to be copied from GPU to CPU. Similarly, to pass a CPU variable to a GPU kernel, the variable has to be copied into GPU.

Each thread in a kernel has access to many kinds of memory, each with different access speed and size, and it is the programmer's responsibility to structure the program in a way that maximal throughput is achieved. At the top level, all running threads have access to global memory. Access to global memory is very slow and read/write operations should be minimized for best performance. Still, the use of global memory is inevitable because the input data has to be stored somewhere during kernel launch. Because of this, CUDA devices enable some optimization for the developer. One of the fundamental

access patterns is coalesced read and write. When a warp accesses memory, the device groups up the operations into one or more transactions depending on the distribution of the memory addresses accessed [6]. Failing to group addresses can lead to serialization of this transaction and can degrade the performance of the program.

Every block has a faster local memory labeled as shared. This on-chip memory is visible to every thread in the same block and can be used as user-managed data cache [7]. There are two ways to allocate shared memory. The first type is statically allocated shared variables. All these variables are declared using `static __shared__` keywords. The size of each static shared variable has to be known at compile time. The second way is with the use of `extern __shared__` keywords. With the keyword `extern`, the programmer declares that the memory will be dynamically allocated during kernel creation. Dynamically allocated shared memory can be declared only once as an array. To declare multiple variables, the initial array has to be split using type casting and pointer arithmetic. To prevent data race, block synchronization should be performed after writing into shared memory to ensure changes are visible to other threads.

Every thread also disposes with a set of very fast registers visible only to the thread. Data from registers can be shared with other threads in the same warp using `shuffle` functions without the use of shared memory. The number of registers available to the thread is limited by the architecture of the GPU and the compiler tries to minimize the amount of registers used by the block.

2.2.3 Thread Synchronization

The CUDA API enables multiple means to synchronize CUDA threads. The first and most important synchronization is between kernel launches. A kernel launch starts from the CPU and until all threads are done, no other kernel is executed at the same time. This call is nonblocking for the host device, meaning the control is returned right back to CPU, but the work does not have to be executed on the GPU yet. This allows the CPU to push tasks to GPU rapidly and the GPU will perform the tasks in the correct order whenever the hardware is not occupied and ready to be used. To explicitly block the host until all GPU works are done, the programmer has multiple options. One solution is to call `cudaDeviceSynchronize()`. This function waits until all tasks on GPU are done and returns an error code if anything failed, otherwise `cudaSuccess`. Another way to synchronize host and device is to copy memory between these two platforms.

On the lowest level, some CUDA threads are implicitly synchronized on the hardware level due to the execution model. CUDA threads are executed in groups of 32, also known as warp. Warp size is important because during any instance, all threads within warp execute the same instruction (if warps do not diverge because of branching). This approach of execution is labeled as

SIMT (single instruction multiple threads) architecture. When warps diverge, it is necessary to stop the warps that take a different branch path and only allow a subset of threads to execute the instruction. This divergence leads to a performance penalty and diverging threads should be kept to minimum.

To implicitly synchronize threads, the CUDA programming model provides a simple barrier function implemented as `__synchthreads()`. It is important to note that this function can only synchronize threads within the same CUDA block. Upon calling this function, the thread is blocked and has to wait for all the other threads in the same CUDA block to call `__synchthreads()` as well. Once all threads have called the function, all blocked threads are released.

The use of synchronization in a block is to ensure that all critical sections are handled properly and shared memory is properly updated and can be used by other threads, i.e. the data that a thread holds need to be given to another thread. The CUDA API also supports warp `shuffle` operations. These sets of functions allow active threads in a warp to exchange data with the use of registers and are available to devices with compute capability 3.x or higher [6]. With this, data can be moved between threads in a warp without having to stop other threads with synchronization and this approach is generally much faster than writing into shared memory and then calling `__synchthreads()`.

If multiple CUDA blocks need to be synchronized, the best option is to store the data on hand into global memory, end the kernel and synchronize through CPU. Lastly, start a new kernel to continue where the last kernel stopped. This solution works on all CUDA enabled GPUs but on newer devices, the CUDA API introduced the concept of `cooperative groups` [6]. This feature allows the programmer to synchronize threads across blocks without having to end the kernel and inside a CUDA block, it provides a mean to synchronize threads with finer granularity than `__synchthreads()`. The only drawback is that this feature is only available on GPUs supporting CUDA 9 and higher. As such, this feature was not used in the implementation and only the basic synchronization means were used.

To ensure that operations on global memory are executed deterministically, atomic operations were used. To demonstrate how they work, only `atomicAdd` will be described. There are also other atomic operations, but they all work similarly. The function `atomicAdd` takes a memory address and a value as parameter. It atomically adds the value to the address and returns the old value that lied in memory. When two threads add to the same address at the same time, the operation is performed in sequence, meaning one of the threads will have to wait until the first thread finishes the operation before `atomicAdd` can be executed again. The use of atomic operations should be minimized to gain the best possible performance.

2.2.4 CUDA dynamic parallelism

CDP or CUDA dynamic parallelism is an extension of the CUDA API that allows the programmer to launch new kernels within the kernels themselves. This approach is only available to devices with compute capability of 3.5 and higher [6]. By default, only two levels of kernel launch recursion are allowed and to enable deeper recursion, max depth has to be explicitly set on host before kernel launch. With each recursive kernel launch, more memory has to be set aside for parent-child synchronization. Because of this limitation, most algorithms using CDP are memory bound.

2.3 TNL

TNL is an open-source library that helps with the development of efficient numerical solvers. The library is mainly written in C++ and uses templates and other tricks to give the programmer a user-friendly interface. TNL also provides support for modern hardware, such as multicore CPUs, GPUs, and distributed systems. One of the core concepts of TNL is unifying the interface of data structures and algorithms for different memory spaces [1].

2.3.1 TNL data structures

TNL contains some very useful structures to store data, one of which is `TNL::Containers::Array`. The class allows the user to create array-like structures to store information without having to implement memory allocation and deallocation. To create data in GPUs, the memory has to be allocated using CUDA API. This can lead to a lot of coding just to allocate memory, initialize and then deallocate at an appropriate time. Furthermore, every GPU operation has to be checked as failed function calls from CUDA API are not caught explicitly. All allocations and deallocations are handled in the constructor and destructor of the object using RAII [1] and only a high level interface is accessible to the user.

The `Array` class exposes some public methods allowing data reading and modification. To access memory, `Array::getElement` can be used. This method guarantees that the element is returned even if called from the wrong address space (i.e. `Array` holds data in GPU and the CPU fetches the data). Similarly, `Array::setElement` can be used to modify an element and the method guarantees correctness, no matter which device called the method.

`Array` also implements `operator[]`, allowing the developer to access to the elements as if it were a regular array of elements. The operator returns a reference to the element, as such, calling the method from host for data allocated on device (or vice versa) leads to segmentation fault (on the host system) or broken state of the device [1].

```
1  ArrayView<int, Device> view = ...;
2  auto addLambda = [=]__cuda_callable__(int i, int j)
3                   {return view[i] + view[j];}
```

Listing 1: Example of a lambda capturing a View. The lambda adds two values from Array.

2.3.2 TNL View structures

To create efficient algorithms, memory operations — such as copying — should be kept to minimum. To move `Array` around, there are two efficient solutions. One is to capture the object by reference. This approach works well when the object is needed in the same device, but when launching a kernel, all parameters need to be captured by value. Another problem arises when `lambdas` are used. To create a `lambda` callable in kernel, all variables need to be captured by value. This leads to a lot of unnecessary copying of data and the performance is greatly degraded. All these problems stem from the fact that CPU memory and GPU memory are separate and referencing data from the wrong memory space can lead to failure of the program.

The other solution, and the one used in TNL, relies on binding the internal pointer and wrapping it in another class. This shallow copy of `Array` is labeled as `ArrayView` and implements the same methods as the original class. Creation and copying of `ArrayView` is very fast because only the pointer and the size of the array are passed around.

2.3.3 Lambdas

CUDA kernels support the use of C++ lambdas and these user defined functions can be used in many general templated functions. CUDA lambdas, to be callable from GPU, need to have `__device__` specifier. The TNL library unifies both `__host__` and `__device__` specifiers under the `__cuda_callable__` macro. To give an example, in Listing 1, the piece of code creates a lambda that adds two elements from a view. Lambdas created in CPU and called in GPU also can not capture variables by reference because of difference in memory address space, as such all variables need to be captured by value.

2.4 Notation

In this work, the goal of the algorithms is to sort a sequence. A sequence a containing $(a_1, a_2, a_3 \dots a_n)$ will be used as $a[1 \dots n]$. Element at position i is denoted as a_i. To concatenate sequences $a[i \dots j]$ and $a[k \dots l]$, $a[i \dots j, k \dots l]$ will be used.

For algorithms, this thesis will also provide information about how long the implementation will run. The standard notation for time and space complexity is the asymptotic notation [8] and this notation will also be used in this thesis. An algorithm that needs *at most* n operations that take constant time is denoted as $O(n)$. If exactly $n \log n$ operations are needed — $\Theta(n \log n)$. $\Omega(n^2)$ will be used when the algorithm needs *at least* n^2 operations.

2.5 Sorting problem

In this thesis, the goal is to sort data of arbitrary type. As such, only algorithms using a binary operator $<$ will be considered. The comparison operator is a function that answers for every two elements a_i, a_j whether $<$, $>$ or $=$ is true [8].

The sorting problem is about reordering the input in such a way that $a_i < a_{i+1}$ or $a_i = a_{i+1}$ for $1 \leq i < n - 1$. There are many types of sorting algorithms and each has a different property. As stated in [8], the most notable property that a sorting procedure can have is:

Definition 2.1. Let a_i, a_j are equal. Also, let $a_{i'}$ is the new position of a_i and $a_{j'}$ is the new position of a_j after sorting. A sorting algorithm is stable if $i \leq j$ then $i' \leq j'$.

Definition 2.2. A sort algorithm is in-place if the amount of memory needed to move the elements is constant. An auxiliary array to store the elements temporarily is allowed, but only a constant number of elements can be outside the input array.

2.5.1 Single thread limitation

For comparison-based sorting algorithms, the best deterministic implementation will work in time $\Omega(n \log n)$ [8] for the worst input. To gain any real speed-up, there are a few options that come to mind. One solution is to implement an algorithm that works in time $\Theta(n \log n)$ but has a small multiplicative constant. Another solution is to use a faster CPU that can execute instructions faster. Both of these solutions are still limited by the theoretical time complexity.

To gain theoretical speed-up, at least one of the constraints needs to be broken. One of which is using a non-comparison based sorting algorithm. A representative of such algorithm is Radix sort or Counting sort [8]. These solutions are not ideal because they cannot sort non-numeric values.

The solution introduced by this thesis breaks the deterministic constraint and parallelization of the algorithm will be done to sort faster than $O(n \log n)$.

2.5.2 Overview of existing algorithms

For CPU, many efficient sorting algorithms have been implemented and almost any numerical or standard library contains a sort function. For C++, the standard is `std::sort` from the `libstdc++` library. The sorting algorithm selected in the library comprises of two algorithms — Introsort is used on a big input and then the algorithm switches to Insertion sort once the input is small enough. To give an example for other languages, Java 7, Python, Swift, and Rust, to name a few, implement Timsort [9]. This algorithm uses Merge sort and Insert Sort. As shown, these highly optimized sorts in the libraries all have one thing in common, in the first phase, a more complex method is used to preprocess a big input and then a simpler procedure is used to finish the sorting on smaller subsequences.

For parallel implementation, the goal of the algorithm is not only to have a good time complexity but a big bottle neck comes from how memory is accessed. A major factor that influences overall time needed to sort comes from the limited bandwidth of the GPU [10]. The main optimizations that can be done to improve run-time usually are based on memory access patterns and efficient use of faster memory. This is also recommended by the CUDA Programming Guide [6] as the most important aspect to gain speed-up.

The CUDA toolkit comes with a library supporting various operations on GPU called `thrust` [5]. This templated library contains a sorting function that can be run on either CPU or GPU. Internally the function implements Radix sort, one of the most popular sorting algorithm for GPU [10]. Another popular parallel sorting algorithm is Bitonic sort [11]. Bitonic sort belongs to the family of so called *sorting network* algorithms. Algorithms from this group are easy to implement as it is possible to map the network directly to a GPU kernel [10]. A very natural approach to parallelizing sorting involves reducing a big problem to smaller problems and recursively (and in parallel) process newly created buckets. This solution was presented by Cederman et al. in their GPU-Sort paper [3] that uses Quicksort to partition a sequence in parallel on GPU. Later on, Manca et al. [4] published their optimization of GPU-Sort for CUDA devices.

In this thesis, Quicksort and Bitonic sort will be implemented. Both of them are comparison based algorithms that only need a comparator defined. This allows for the data type to be of arbitrary type.

Chapter 3

Bitonic Sort

3.1 The Bitonic sort algorithm

Bitonic sort was first introduced by K. E. Batcher in 1968 [11]. It is a comparison-based sorting algorithm that uses multiple layers of sorting networks to sort the whole input sequence. To understand the algorithm, it is necessary to define the following terms.

Definition 3.1. A sequence is bitonic if it is a concatenation of two monotonic sequences, one ascending and the other descending. Furthermore, the sequence is still bitonic, even if we split the sequence anywhere and interchange the sequences.

Definition 3.2. Bitonic merge is an operation that transforms a bitonic sequence into a monotonic sequence.

With these definitions, the sorting process is simple: split the input array in two parts, sort one part recursively in one direction and the other part in a different direction. Because the two parts are next to each other, a concatenation operation is not needed and by definition, the input is now bitonic. Then Bitonic merge is used to transform the bitonic sequence into a monotonic one and the input is sorted. The operation is called Bitonic merge because it merges two monotonic sequences and creates a bigger monotonic sequence.

Example 3.1. Any sequence of length 1 is bitonic, additionally any sequence of length 2 is also bitonic. The first element creates an ascending monotonic sequence of length 1 and the second element creates a descending monotonic sequence.

Furthermore, any sequence of length 3 is also bitonic. Any two neighbouring elements create a monotonic sequence and the leftover third element can be either ascending or descending depending on the need.

Example 3.2. $a = (1, 2, 4, 5)$ is a bitonic sequence. The subsequence $(1, 2, 4, 5)$ is ascending and the descending part is of length 0. Another way to split the input can be $(1, 2, 4)$ and (5) where the first part is ascending and the second part is descending.

Example 3.3. $a = (3, 4, 6, 5, 0, 2)$ is a bitonic sequence. First, split the sequence a into $(3, 4, 6, 5)$ and $(0, 2)$ and interchange them, creating $a' = (0, 2, 3, 4, 6, 5)$. The sequence a' is bitonic which means the original a was also bitonic.

Example 3.4. $a = (3, 4, 6, 5, 0, 5)$ does not create a bitonic sequence.

3.1.1 Bitonic merge

Theorem 3.1. Let $a[1 \dots 2n]$ be bitonic sequence. Let $d_i = \min(a_i, a_{i+n})$ and $e_i = \max(a_i, a_{i+n})$ for $1 \leq i \leq n$. Then both $d[1 \dots n]$ and $e[1 \dots n]$ are bitonic. Also, $\max(d[1 \dots n]) \leq \min(e[1 \dots n])$. We call this operation Bitonic split.

The merging operation is described in [11]. To transform a bitonic sequence into monotonic, Bitonic split is repeatedly performed on the input. This operation splits the input into two bitonic sequences, one holding n smaller elements while the other holds n bigger elements. This operation is then recursively applied to each of the newly created bitonic sequences until the input is of size 1. The original input is now monotonic.

To prove Theorem 3.1, some definitions and lemmas will be needed first.

Definition 3.3. A left cyclic shift is an operation that reorders the input sequence $a[1 \dots n]$ into $a[2 \dots n, 1]$. Similarly, a right cyclic shift moves all elements by 1 to the right, creating $a[n, 1 \dots n-1]$.

Formally, left cyclic shift can be written as

$$a'_i = \begin{cases} a_{i+1} & i < n \\ a_1 & i = n \end{cases}$$

and right cyclic shift as

$$a'_i = \begin{cases} a_n & i = 1 \\ a_{i-1} & i > 1 \end{cases}$$

Remark 3.1. Splitting a sequence of length $2n$ into 2 sequences of length $|k|$ and $|l|$, such that $2n = |k| + |l|$, and interchanging them can be seen as performing left cyclic shift $|k|$-times or right cyclic shift $|l|$-times.

Lemma 3.1.1. *Performing a right cyclic shift on $a[1 \dots 2n]$ also cyclically shifts $d[1 \dots n]$ and $e[1 \dots n]$. The sequences undergo the same change when left cyclic shift is applied.*

Proof. For every element a_i, there exists exactly one other element that is n positions away (meaning the paired element is a_{i+n} or a_{i-n}). The pair is not broken up even after cyclically shifting $a[1 \dots 2n]$. □

Therefore, it is possible to cyclically shift the input until

$$a_1 \leq a_2 \leq \dots \leq a_j \geq a_{j+1} \geq \dots \geq a_{2n}$$

a_j is the global maximum of the input and $a[1 \dots j]$ is increasing and $a[j \dots 2n]$ is decreasing. This satisfies the bitonic property and does not change the maximum value of $d[1 \dots n]$ nor minimum of $e[1 \dots n]$.

Lemma 3.1.2. *Flipping the input $a[1 \dots 2n]$ into $a[2n \dots 1]$ does not affect the bitonic property. After flipping, $d[1 \dots n]$ becomes $d[n \dots 1]$ and $e[1 \dots n]$ is also flipped.*

Using Lemma 3.1.2, it is sufficient to prove Theorem 3.1 for case $a_1 \leq a_2 \leq \dots \leq a_j \geq a_{j+1} \geq \dots \geq a_{2n}$, where $n < j \leq 2n$.

Now with all the necessary conditions set, Theorem 3.1 can be proven.

Proof. Let $a[1 \dots 2n]$ be the bitonic sequence that needs to be merged by Bitonic merge. There are 2 cases that can happen.

Let $a_n \leq a_{2n}$, then for $1 \leq i \leq n$, $a_i \leq a_{i+n}$ so $d_i = \min(a_i, a_{i+n}) = a_i$ and $e_i = \min(a_i, a_{i+n}) = a_{i+n}$. $d[1 \dots n] = a[1 \dots n]$ and it has been proposed that the sequence is ascending in this part, therefore $\max(d[1 \dots n]) = d_n = a_n$. $e[1 \dots n] = a[n+1 \dots 2n]$, furthermore, $\min(e[1 \dots j-n]) = e_1 = a_{n+1}$ because this part is increasing and $\min(e[j - n \dots n]) = e_n = a_{2n}$ as this part is decreasing. In total, $\min(e[1 \dots n]) = \min(e_1, e_n) = \min(a_{n+1}, a_{2n})$.

Together $\max(d[1 \dots n]) = a_n \leq \min(a_{n+1}, a_{2n}) = \min(e[1 \dots n])$. If a_{n+1} is minimum then the inequality is still true because $a[1 \dots j]$ is increasing, otherwise a_{2n} is minimum and the inequality is also true because of the initial proposition.

Let $a_n > a_{2n}$, then there exists such a $k : j \leq k \leq 2n$, that $a_{k-n} \leq a_k$ and $a_{k-n+1} > a_{k+1}$. This stems from the fact that $a[j-n \dots j]$ is increasing while $a[j \dots 2n]$ is decreasing. d_i and e_i are created as follows:

$$\left.\begin{aligned} d_i &= a_i \\ e_i &= a_{i+n} \end{aligned}\right\} \text{for } 1 \leq i \leq k-n$$

and

$$\left.\begin{aligned} d_i &= a_{i+n} \\ e_i &= a_i \end{aligned}\right\} \text{for } k-n < i \leq n$$

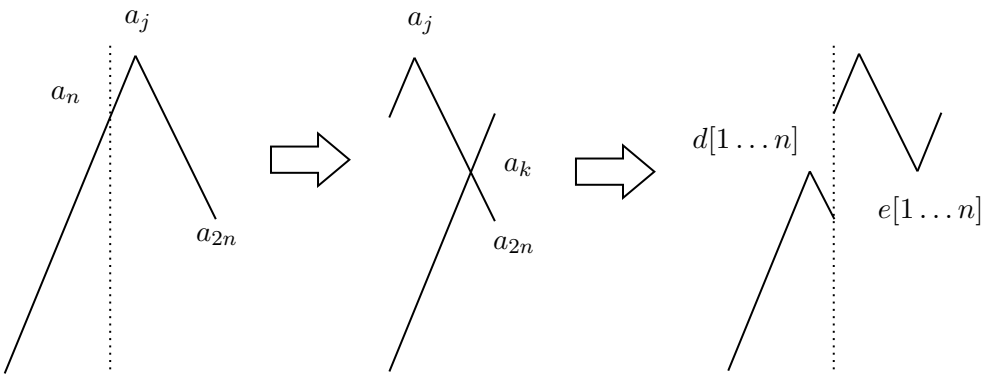

Figure 3.1: One step of Bitonic merge where $a_n > a_{2n}$.

The first part $d[1 \dots k-n] = a[1 \dots k-n]$ is ascending and the second part $d[k-n+1 \dots n] = a[k+1 \dots 2n]$ is descending, this satisfies the bitonic property. $e[1 \dots j-n] = a[n+1 \dots j]$ is ascending, $e[j-n+1 \dots k-n] = a[j+1 \dots k]$ is descending and $e[k-n+1 \dots n] = a[k-n+1 \dots n]$ is ascending. Let us also notice that it is possible to perform right cyclic shifts on $e[1 \dots n]$ to create $e[k-n+1 \dots n, n+1 \dots k-n] = a[k-n+1 \dots k]$ which is in part ascending until $e_{j-n} = a_j$ and then descending till the end. Sequence $e[1 \dots n]$ therefore also satisfies the bitonic property.

After analyzing, $\max(d[1 \dots n]) = \max(d_{k-n}, d_{k-n+1}) = \max(a_{k-n}, a_{k+1})$ and $\min(e[1 \dots n]) = \min(a[k-n+1 \dots k]) = \min(a_{k-n+1}, a_k)$. If $d_{k-n} = a_{k-n}$ is the maximum, then $a_{k-n} \leq a_{k-n+1}$ because this part is ascending and $a_{k-n} \leq a_k$ due to initial proposition. Else $d_{k-n+1} = a_{k+1}$ is maximum, then $a_{k+1} \leq a_{k-n+1}$ due to intial proposition and $a_{k+1} \leq a_k$ because the sequence is descending in this part. In all 4 cases, the inequality holds. □

3.1.2 Sorting in-place

To split the input, the sequences $d[1 \dots n]$ and $e[1 \dots n]$ do not need to be allocated. It is sufficient for $d[1 \dots n] = a[1 \dots n]$ and $e[1 \dots n] = a[n+1 \dots 2n]$. With this, it is only necessary to compare a_i and a_{i+n} and swap the elements if needed.

Formally $d_i = a_i = \min(a_i, a_{i+n})$, $e_i = a_{i+n} = \max(a_i, a_{i+n})$ for $1 \leq i \leq n$. Here the min function uses the $<$ comparator to determine which element is smaller, similarly max returns the bigger element of the two. These 2 operations can furthermore be optimized to compare $a_{i+n} < a_i$ first and only swap them if they are in the wrong order.

3.1.3 The recursive algorithm

The resulting algorithm can be seen in Algorithm 3.1.1 and Algorithm 3.1.2. To sort the input, `BitonicSort` is recursively called twice, once on $a[1 \dots n]$ and the second time on $a[n+1 \dots 2n]$. It should be noted that one of the sequences needs to be sorted in the opposite direction in order for them to create a bitonic sequence at the end.

The last operation that needs to be done is `BitonicMerge`. The function implements Bitonic split — moves n smaller elements into the first half and n bigger elements into the second half. The two resulting subsequences are also bitonic. Then `BitonicMerge` is called on both the subsequences.

Algorithm 3.1.1: Recursive Bitonic sort

Input: array $a[1 \dots 2n]$, sort ordering *ordering*

```
1  if 2n = 1 then
2      return
3  BitonicSort(a[1...n], opposite ordering)
4  BitonicSort(a[n+1...2n], ordering)
5  BitonicMerge(a[1...2n], ordering)
```

Algorithm 3.1.2: Recursive Bitonic Merge

Input: array $a[1 \dots 2n]$, sort ordering *ordering*

```
1  if 2n = 1 then
2      return
3  for i from 1 to n do
4      if ordering is ascending then
5          if a_{i+n} < a_i then
6              swap(a_i, a_{i+n})
7      else if ordering is descending then
8          if a_i < a_{i+n} then
9              swap(a_i, a_{i+n})
10 BitonicMerge(a[1...n], ordering)
11 BitonicMerge(a[n+1...2n], ordering)
```

3.1.4 Time complexity

Bitonic split creates the sequences $d[1 \dots n]$ and $e[1 \dots n]$. For each comparison, two elements from $a[1 \dots 2n]$ are processed so a Bitonic split is of time $\frac{n}{2} = \Theta(n)$. Bitonic merge performs one Bitonic split and then recursively

calls itself twice (once on $d[1 \dots n]$ and once on $e[1 \dots n]$). From this, we get $T(n) = \frac{n}{2} + 2T(\frac{n}{2})$, using Master theorem, the result is $T(n) = \Theta(n \log n)$.

To sort the whole sequence, first the input is split in half and Bitonic sort is called on each half, then merge operation is called. The complexity is $T(n) = 2T(\frac{n}{2}) + n \log n = \Theta(n \log^2 n)$ [12].

3.1.5 Sorting not aligned sequences

So far, the Bitonic sort that has been introduced could sort only sequences of length $n = 2^m$. To sort sequences of any length, the algorithm has to be modified. There is the trivial solution [13] of extending the sequence to the length of the next closest power of 2 and fill the missing elements with max-value (min-value when sorting in descending order). However, such a solution is not very efficient for big inputs. First, a new temporary array of length $2^k \geq n$ has to be allocated, the whole input has to be copied into auxiliary memory, sorted there and then copied back without padding elements. With temporary memory, the option to sort in place is lost and the space complexity becomes $\Theta(n)$. However, for smaller elements this approach might be more efficient due to low overhead compared to the next introduced solution.

The Bitonic merge operation only requires the input to be bitonic and it does not matter whether the sequence is a concatenation of an ascending and then descending sequence or descending and then ascending sequence. Using this, the change is as follows [13]: let $n \geq 1$ be the length of input and k is the biggest power of 2 that still smaller than n (formally $k < n \leq 2k$), then split the original sequence into $a[1 \dots k]$ as left part and $a[k+1 \dots n]$ as right part. Sort the left part in descending order and the right part in ascending order. Lastly, call Bitonic merge on the newly created bitonic sequence.

Let us notice that the left part is of size k which is a power of 2. Sorting this part can proceed as normal. For the right part, whenever a_i — where $i > n$ — is needed, a max-value (if sorting in descending order, then min-value is used) can be substituted in. In this situation, no swap is needed and the max-value does not have to exist physically in memory. This approach is called virtual padding. With virtual padding, only the direction of sorting has to be minded during calculation and the algorithm does not lose the property of being in-place nor is the time complexity degraded.

3.2 Parallel algorithm

To implement an efficient parallel version of Bitonic sort, the algorithm has to be modified a little. Instead of using recursion, it is better to build the monotonic sequences in iterations [11]. The algorithm first creates monotonic sequences of length 1, this is trivial. Now the input $a[1 \dots n]$ consists of n monotonic sequences. In the next step, Bitonic merge is used on two neighbouring monotonic sequences, each with length k. This creates new monotonic

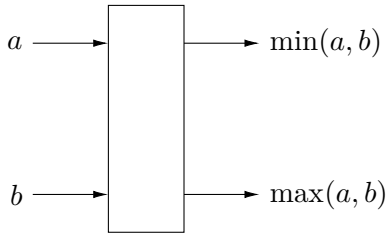

Figure 3.2: Sorting network comparator for two elements, adopted from [11]. The comparator sorts in ascending order.

sequences of length $2k$. To be able to use Bitonic merge, the two monotonic sequences had to be sorted each in different directions (so they create a bitonic sequence). It is also necessary to merge the correct way (ascending or descending) so that the next step will have two oppositely sorted sequences to merge again.

3.2.1 Sorting network

Bitonic sort, when used iteratively, can be represented as a sorting network. A sorting network, as defined in [14], is a hardware sorter circuit. The basic element of a sorting network is a comparator. A comparator has two input lines and two output lines. It receives two numbers on its input lines, compares them, and outputs the maximum on its higher output line and the minimum on its lower output line. By interchanging the output lines, a comparator that sorts in descending order can be gained. A simple scheme of a comparator is shown in Figure 3.2.

To sort the input, the comparators need to be connected in such a way that they permutate the input correctly and send the elements in the correct order to the output. The sorting network in Figure 3.3 shows a way to connect comparators to simulate Bitonic sort. The network consists of multiple layers which are called steps and multiple steps are grouped into phases. The strength of sorting networks lies in the fact that it can be implemented on the hardware level to run parallel. Bitonic sort due to its recursive property is very convenient for mass production of the hardware. A large bitonic sorter can be created by connecting smaller Bitonic sorters.

Example 3.5. In this example, only phase 2, step 1 and 2 of Bitonic merge will be demonstrated. The steps will be described in text and a general scheme with a bitonic sorter will be shown in Figure 3.4.

Let input $a[1 \dots n] = ((6, 5), (3, 7), (10, -3), (5))$ be a concatenation of 4 monotonic sequences.

Setup: The current monotonic length is 2 (the last part (5) is an exception as the input is not aligned into power of 2). In the beginning, concatenate two neighbouring monotonic sequences to create a bitonic sequence twice as

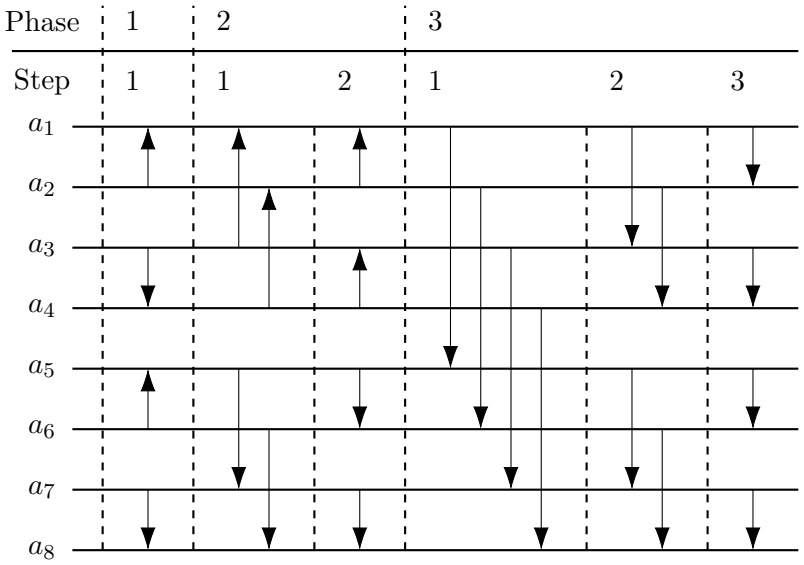

Figure 3.3: Bitonic sorting network for 8 elements, adopted from [13].

big. $((6,5),(3,7))$ is bitonic and Bitonic split can be applied. The same thing happens to $((10,-3),(5))$. The sequence $((10,-3),(5))$ is not aligned and needs to sorted in ascending order, this forces the left part $((6,5),(3,7))$ to be sorted in descending order.

Phase 2, step 1: The elements 6 and 3 are compared first, maximum is sent through the top line and minimum though bottom line, same with 5 and 7. This creates the sequence $(6,7,3,5)$. Here $d[1\ldots k] = (6,7)$ and $e[1\ldots k] = (3,5)$. At the same time, 10 and 5 are compared and swapped as they are in the wrong order. -3 does not have a partner and is not swapped. After Bitonic split, the input becomes $(6,7,3,5,5,-3,10)$. It should be noted that all comparisons and swaps done in this step were independent and can be performed in parallel.

Phase 2, step 2: Each bitonic sequence is now of length 2. To demonstrate, the input can be split as $((6,7),(3,5),(5,-3),(10))$. Elements 6 and 7 are still part of a sequence that needs to be sorted in descending order and therefore are in wrong order and need to be swapped. Same with $(3,5)$. $(5,-3)$ needs to be sorted in ascending order and are also swapped. (10) does not have a partner and is not swapped.

The result then turns into $((7,6,5,3),(-3,5,10))$ and the sequence is ready for another application of Bitonic merge which will turn the input into 1 monotonic sequence.

As demonstrated in Example 3.5, a Bitonic merge phase consists of multiple Bitonic splits. All swaps in one Bitonic merge step are independent and can be performed in parallel.

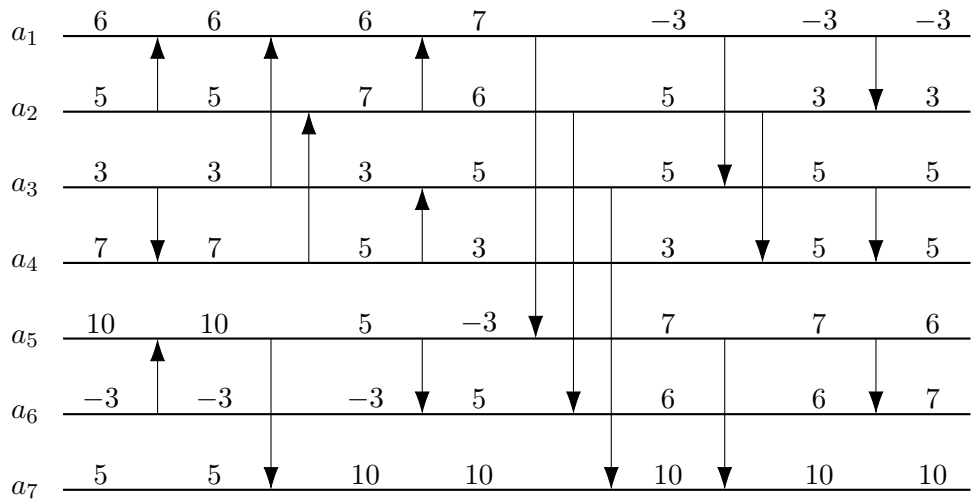

Figure 3.4: Sorting example 3.5 with a Bitonic sorting network.

3.2.2 Time complexity of parallel implementation

If there are $O(n)$ threads available to work during each step, it is possible to perform all $\lceil \frac{n}{2} \rceil$ swaps of a Bitonic split in $O(1)$ as all comparisons are independent. The time complexity then collapses to the number of phases and steps. After the first phase, each part is a sorted sequence of length 2. After second phase, the length is 4. After k phases, each part is a monotonic sequence of length 2^k. Thus, the number of phases is $O(\log_2 n)$. The first step of phase i will split a bitonic sequence of length 2^i into 2 bitonic sequences, each with length 2^{i-1}. Bitonic split is then applied on each of the newly created bitonic sequences until they are of length 1. Therefore, each phase needs $\log_2 2^i = i$ steps.

Together, the complexity of bitonic sort is

$$\sum_{i=1}^{\log_2 n} i = \log_2 n \frac{1 + \log_2 n}{2} = O(\log^2 n)$$

3.3 Existing implementations

Bitonic sort for GPU has been studied intensively due to its simplicity [10] and has been implemented by many researchers. Despite this, most of them are implemented to sort only integer types and can only handle aligned sequences. A very good and optimized implementation of Bitonic sort is available in the CUDA SDK [2], but this implementation can only sort unsigned integer types and the interface forces the user to sort key-value which greatly reduces the performance when keys are also values. The goal is to implement a Bitonic

```
1 ArrayView<double, Device::Cuda> view = ...;
2 auto Fetch = [=]__cuda_callable__(int i){return view[i];};
3 auto Cmp = [=]__cuda_callable__(const double & a,
4                                 const double & b)
5                 {return a < b;};
6 auto Swap = [=]__cuda_callable__(int i, int j) mutable
7                 {TNL::swap(view[i], view[j]);};
8 bitonicSort(0, view.getSize(), Fetch, Cmp, Swap);
```

Listing 2: Example of fetch and swap version of Bitonic sort with lambdas.

sort that can be called both from CPU and GPU. The function will not be restricting, meaning the data type will be templated and there will be no restriction on the size of the input.

3.4 Implementation of Bitonic sort with CUDA

On the highest abstract level, there are two available interfaces of Bitonic sort. The main one is `BitonicSort(ArrayView<Value, Device>)`. This function takes an `ArrayView` as input and will sort the input using Bitonic sort on GPU. As default, the comparator for two elements from the input is `operator<` and as such will sort in ascending order. Same as with `std::sort`, an overloaded version of this function with custom comparator is available. It should be noted that the comparator has to be callable from device, because of this, the function or lambda has to have `__device__` specifier. With a different comparator, it is possible to sort any data type ascending or descending.

To sort key and value types of data, the programmer has two options: to zip the key and value into one structure and then use a custom comparator, or to use the `BitonicSort` interface with fetch and swap. The second approach allows the user to sort any type of indexed container (arrays, vectors, lists etc.) in-place without having to create additional structure to zip key-value. The function needs five values — first and last index of the structure that needs to be sorted, a fetch function that returns the element on i-th position, a compare function that compares two elements and a swap function that swaps elements on i-th and j-th position. The swap function allows the user not only to swap the fetched elements but also perform other operations (such as swap values in another structure or do other calculations). Because `swap` takes non `const` values as argument and changes memory, if used as lambda, the keyword `mutable` has to be used, on top of `__device__` specifier. This approach is universal for almost any data container, but the performance is greatly reduced because all operations have to be carried out in global memory. An example on how to call the function is provided in Listing 2.

3.4.1 Host side

To sort the input, Bitonic merge has to be repeatedly called, each time with different parameters. In section 3.2, the algorithm was described and pointed out how many times Bitonic merge needs to be called during i-th phase. The Algorithm 3.4.1 shows that the iterative approach can be easily implemented with two nested `for` loops.

Algorithm 3.4.1: Bitonic sort kernel launch

Input: array $arr[1 \dots n]$

```
1 gridSize, blockSize ← calculate optimal configuration
2 for i ← 1 to ⌈log n⌉ do
3 |   for j ← i to 1 do
4 |   |   BitonicMerge_{i,j}<gridSize, blockSize>(arr[1 ... n])
```

The host side is mainly used to synchronize kernels inbetween kernel launches. Each kernel launch will start the swapping of a layer of a Bitonic sorting network. It is assumed that the number of threads available in GPU is at least half the size of the input sequence. Each block will be started with 512 or 256 threads and the theoretical maximum number of blocks that can be launched is $2^{31} - 1$ [6]. To use up all threads, the input would have to have more than 2^{40} elements and it is not assumed that such a big input will fit in the memory of a GPU.

In host, `blockSize` and `gridSize` is calculated and these two numbers are used for every kernel launch until the input sorted. Each CUDA thread is implemented to compare two elements and simulate a comparator in a step of a sorting network, as such, to sort a sequence with n elements, $\lceil \frac{n}{2} \rceil$ threads will be needed in total.

To maximize shared memory usage, the number of threads per CUDA block is selected as 512 or 256. By default, 512 is used and 1024 elements are processed in every CUDA block, but if the data type is too large (some complicated structure), 256 can be used to reduce the memory consumption per block. If even 256×2 elements cannot be copied into shared memory, then all operations will be executed in global memory with 512 threads per block.

3.4.2 Device side

Each CUDA thread in Bitonic sort is implemented to simulate a comparator from Bitonic sorting network. The kernel consists of calculating which two elements need to be fetched, calculating whether the two elements need to be sorted in ascending or descending order and the last operation that needs to be done is comparing them and then execute swap on the elements if needed.

```
1  int i = blockIdx.x * blockDim.x + threadIdx.x;
2  int offset = bitonicLen / 2;
3  int s = (i / offset) * bitonicLen + (i % offset);
4  int e = s + offset;
```

Listing 3: Implementation of how the two compared elements by thread `i` — a_s and a_e — are calculated.

To calculate which two elements need to be accessed by a thread, the *global* thread id needs to be known and the current step and phase of the Bitonic merge. In the implementation, one of the parameters passed to the kernel is `bitonicLen`, this value indicates how long each bitonic sequence is in the input and directly corresponds to the phase and step of the Bitonic sort. Global id of the thread can be calculated (as shown in Listing 3) with just CUDA available thread variables (`blockDim, blockIdx, threadIdx`) and is labaled as `i`. These two values are then used to calculate the first element that will be used. The first element is on position `s` and the second element is on `e = s + bitonicLen/2`. These two values correspond to $a_s = a_i$ and $a_e = a_{i+n}$ introduced in the theoretical part.

3.4.3 Calculating the direction of swap

A common operation that every thread calculates is whether the two compared elements should be in ascending or descending order. After *completing a phase*, the order of every two neighbouring monotonic subsequences should be alternating so they can create a bitonic sequence. In the implementation, it has been decided that every even subsequence will be sorted in descending order and every odd subsequence sorted in ascending order. The only exception is for the last two subsequences. As discussed in subsection 3.1.5, the implementation will use the version with virtual padding and it is important to sort the last subsequence — which can potentially be not aligned — in ascending order.

3.4.4 Optimizations

To optimize Bitonic sort, shared memory will be used. Nevertheless, the use of shared memory is not the only optimization that can be done to gain speed-up. One of the tricks used in the implementation is based on bitwise operations. The modulo operator is very slow in GPUs and can be replaced with faster bitwise `&` under some circumstances. Whenever `(i mod m)` is calculated and `m` is a power of 2, the whole operation can be replaced with `i&(m-1)` [6].

Instruction change is not the only optimization that can be used in GPU. Another big improvement can be gained by reducing divergent threads in a warp. For this, a faster version of compare and swapped is used (Listing 4).

```
1  template <typename Value, typename CMP>
2  __cuda_callable__
3  void cmpSwap(Value &a, Value &b,
4               bool ascending, const CMP &Cmp)
5  {
6      if (ascending == Cmp(b, a))
7          TNL::swap(a, b);
8  }
```

Listing 4: Implementation of fast compare and swap with templated parameters.

The comparison consists of checking the direction of swapping that is required and calling the `Cmp` function on two elements. Here, `Cmp` is just a lambda that replaces `operator<`. With this implementation, the check for `ascending` does not create a fork and only 1 call of `Cmp` is needed. It should also be noted that `Cmp` is called as `b < a`. This lambda call checks if the two elements are sorted in *descending* order.

3.4.5 Shared memory in Bitonic Sort

Algorithm 3.4.2: Bitonic sort kernel launch with shared memory

Input: array $arr[1 \dots n]$

```
1 gridSize, blockSize ← calculate optimal configuration
2 bitoniSort1stPhase<gridSize, blockSize>(arr[1 ... n])
3 for i ← log2 blockSize + 1 to ⌈log n⌉ do
4     for j ← i to 1 do
5         if 2^j > 2 * blockSize then
6             bitonicMerge_{i,j}<gridSize, blockSize>(arr[1 ... n])
7         else
8             bitonicMergeShared_{i,j}<gridSize, blockSize>(arr[1 ... n])
9             break
```

To maximize the usage of shared memory available to a CUDA block, Bitonic sort is implemented in three parts (Algorithm 3.4.2). The first optimization is used when a whole bitonic sequence can be copied into shared memory. In this case, the only synchronization that will be needed is block-wide synchronization. The function is labeled as `bitonicMergeShared` in the implementation. As seen (Figure 3.5, part b), a CUDA block accesses the same elements multiple times and thus can be processed in shared memory.

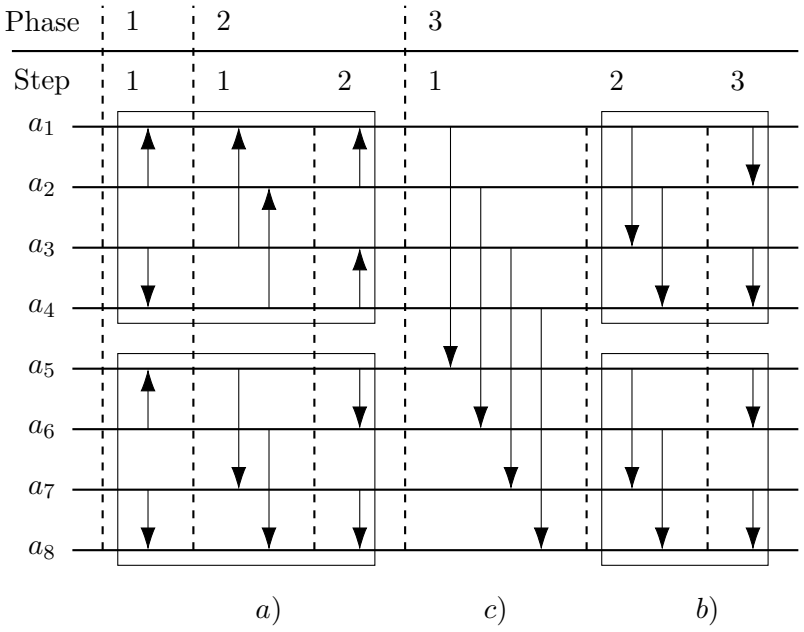

Figure 3.5: Display of shared memory usage for Bitonic sort with 8 elements and 2 CUDA blocks, each block having 2 threads:
a) usage of shared memory in `bitoniSort1stStepSharedMemory`,
b) usage of shared memory in `bitonicMergeShared`,
c) comparison and swapping is done in global memory through `bitonicMerge`.

First, the elements are cached in shared memory, then the elements in shared memory are sorted using repeated application of Bitonic Merge. It is necessary to synchronize threads in the CUDA block after every merge to update the shared memory correctly. Once the elements in shared memory are sorted, they are copied back into global memory.

The second optimization is used right in the beginning. Instead of building the monotonic sequences iteratively, it is possible to create monotonic sequences of length `blockDim.x*2` directly in a CUDA block. Each CUDA block copies a part of the sequence into memory and sorts it completely in shared memory (Figure 3.5, part a). It is necessary for neighbouring CUDA blocks (their `blockIdx` differs by 1) to sort the subsequences in opposite directions so once block sort is done, the neighbouring monotonic sequences can be concatenated into a bitonic sequence. The whole procedure is done in `bitoniSort1stPhase`.

The last function is a regular Bitonic merge that accesses elements in global memory and only simulates one step of Bitonic sort (Figure 3.5, part c).

```
1  template <typename Value, typename CMP>
2  __device__
3  void bitonicSort_Block(ArrayView<Value, Devices::Cuda> src,
4                         ArrayView<Value, Devices::Cuda> dst,
5                         Value *sharedMem, const CMP &Cmp);
```

Listing 5: Interface of block Bitonic sort that uses shared memory. The function is callable directly from a CUDA kernel.

3.5 Bitonic sort from GPU

Bitonic sort was also implemented to be able to sort *from* GPU. These sets of functions were implemented so that the sort procedure could be called directly from a GPU kernel. The device function requires every thread in a block to call the `bitonicSort_Block` function. The function has two overloaded versions. The first one uses shared memory to sort, as such, pointer to shared memory address has to be passed as parameter. It is also necessary for shared memory to be at least as big as the input array because the whole data will first be copied in shared memory, sorted there and then copied back. The last operation copies from shared memory back into global memory. We used this to our advantage and allowed in the implementation to copy the result into another array than the source array. This allow the function interface to be more flexible. The full signature of the function is shown in Listing 5.

The other overloaded version of the function with the same name also allows a CUDA block to use Bitonic sort directly from GPU but will not use shared memory. The Bitonic sort used in this version is in-place and executes all compares and swap in global memory.

Chapter 4

Quicksort

4.1 The Quicksort algorithm

Quicksort is one of the fastest sequential comparison based algorithm in practice with optimal theoretical time complexity on average — $O(n \log n)$[8]. The pseudo code of the algorithm is shown in 4.1.1.

Algorithm 4.1.1: Quicksort algorithm

Input : array $A[1 \dots n]$

```
1 if n ≤ 1 then
2     return
3 pivot ← select an element from A[1...n]
4 i ← partition A[1...n] using pivot
5 quicksort(A[1...i])
6 quicksort(A[i+1...n])
```

This basic idea of Quicksort was initially introduced in [15]. First, a partition procedure is applied on the input sequence. This operation moves all elements smaller than the pivot to $A[1 \dots i]$, all elements greater or equal to pivot into $A[i+1 \dots n]$. There also exist versions of partitioning that split the array $A[1 \dots n]$ into three parts, the left part will hold elements smaller than pivot, the middle part will have elements equal to the pivot, and the right part will contain elements greater than the pivot Then Quicksort is recursively called on the left and then right part. This approach of problem solving where a big input is split into smaller problems is categorized as *divide and conquer* algorithm [8]. The amount of work needed afterwards depends on pivot selection. A general idea of how the memory is partitioned is shown in Figure 4.1.

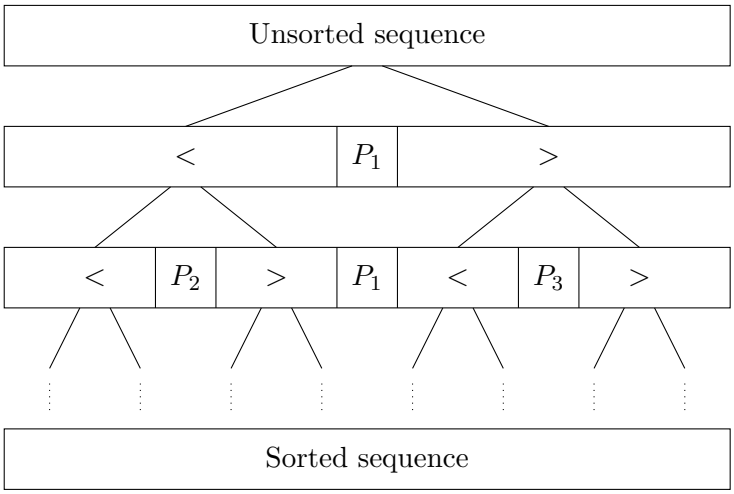

Figure 4.1: Recursive tree of Quicksort

4.1.1 Partitioning algorithms

In Hoare's paper [16], the author suggested using two pointers that go against each other, whenever both elements are in wrong order, they are swapped. The process stops when i and j cross.

Algorithm 4.1.2: Hoare partition scheme

```
   Input : array A, index p, index r
 1 pivot ← A[p]
 2 i ← p − 1
 3 j = r + 1
 4 while True do
 5 |   do
 6 |   |   j ← j − 1
 7 |   while A[j] ≤ pivot;
 8 |   do
 9 |   |   i ← i + 1
10 |   while A[i] ≥ pivot;
11 |   if i < j then
12 |   |   swap(A[i], A[j])
13 |   else
14 |   |   return j
```

Another partition scheme was introduced by Lomuto [17], this method of partitioning also uses two pointers, but this time they start from the beginning of the sequence. At any given iteration, $A[p \dots i]$ will hold elements smaller

Algorithm 4.1.3: Lomuto partition scheme

```
   Input  : array A, index p, index r
1  pivot ← A[r]
2  i ← p − 1
3  for j = p to r − 1 do
4  |  if A[j] ≤ pivot then
5  |  |  i = i + 1
6  |  |  swap(A[i + 1], A[j])
7  swap(A[i + 1], A[r])
8  return i + 1
```

than or equal to the pivot, $A[i+1 \dots j]$ will have elements greater than the pivot. The rest of the array still has not been processed.

The two previously shown implementations of partitioning work in-place and have a time complexity of $\Theta(n)$, this is also the best possible algorithm asymptotically. Every element needs to be compared with the pivot at least once to check if the relative order in memory is correct.

4.1.2 Pivot Choice

With an inappropriate pivot, the load balance between the two newly created sections after partitioning can greatly degrade the performance. In the worst case, only one element gets processed — the pivot — and one of the sections holds the rest of the sequence. After each partitioning, the section that still needs to be sorted gets smaller by 1. This degrades the time complexity of the algorithm to $\Theta(n^2)$.

Ideally, the true median of the input is chosen as the pivot. Then the two sections are of equal size and the recursive tree will have $\log_2 n$ depth. This best case gives the algorithm a $\Theta(n \log n)$ time complexity [8].

In practice, finding the true median is an expensive operation and only an approximation of it is needed. One of the many options on how to choose a pivot is by selecting an element on the same position every time. An example of this approach is choosing the first or last element of the subsequence that needs to be partitioned. This way of selecting pivot will degrade the algorithm to $\Theta(n^2)$ for sorted sequences. In [18], the author suggested picking the pivot as the median of three elements, $A[1], A[\frac{n}{2}]$ and $A[n]$. This choice of pivot is slightly better as it is resistant against sorted sequences.

Another fast way to choose a pivot is by randomly picking an element from the input. It can be proven that the Worst-case Expected-Time of the algorithm is $O(n \log n)$ [19].

4.2 Parallel algorithm

A very natural approach to parallelizing Quicksort is to assign each unsorted subsequence after partitioning to a compute unit and let it sort the subsequences independently in parallel. However, this solution is not optimal during the first few phases. In the beginning, the sequence can be very big and only one thread will not be enough to perform the partitioning. For this reason, not only do the subsequences need to be handled in parallel, but the partitioning procedure also needs to be parallelized.

The partitioning procedure uses a *prefix sum* primitive internally to enable efficient communication between threads. Because of this, prefix sum needs to be described first.

4.2.1 Prefix sum

Prefix sum, or also scan, is an operation that transforms the input $a[1 \dots n]$ into $a'[1 \dots n]$ so that

$$a'_i = \sum_1^i a_i$$

This version of prefix sum is called *inclusive* prefix sum as the element a_i is also included. An *exclusive* prefix sum adds up all previous element but without a_i. Prefix sum can also be defined to use any associative binary operation, but for the implementation, only binary + will be used on integers.

A sequential implementation of prefix sum is simple, for inclusive prefix sum

$$a'_i = \begin{cases} a_i & i = 1 \\ a'_{i-1} + a_i & i > 1 \end{cases}$$

and for exclusive prefix sum

$$a'_i = \begin{cases} 0 & i = 1 \\ a'_{i-1} + a_{i-1} & i > 1 \end{cases}$$

This implementation can not be parallelized as each element has to wait for the result from the previous index. In [20], Blelloch introduced a way to parallelize this operation and reduce the time complexity from linear $O(n)$ to $O(\frac{n}{p} + \log p)$ where n is the length of sequence and p is a fixed amount of processing units available. It should be noted that this only holds if $n > p$, meaning each processing unit has to perform more than one reduction. For $\frac{n}{p} \geq \log_2 p$, the parallel prefix sum operation can be performed in $O(\frac{n}{p})$. If p can be arbitrary, then the time complexity is further reduced to $O(\log n)$ and $\lceil \frac{n}{2} \rceil$ processors will be needed.

Figure 4.2: Scheme displaying, how auxiliary memory will be split after using prefix sum:
(A) part of memory used by threads with $threadId < i$,
(B) part of memory used only by thread i,
(C) part of memory used by threads with $threadId > i$,
(P) pivot(s) position,
(>) part of memory used for elements greater than pivot P.

The parallel version of prefix sum will be used as a subroutine during the partition process to gain even more speed-up. The implementation details will be discussed in subsection 4.4.1

4.2.2 Parallel Quicksort algorithm

In [3, 20], a solution on how to partition in parallel was proposed. With p threads, the input sequence is divided equally between them so that each thread reads $\frac{n}{p}$ elements. Each thread then computes amount of smaller and bigger elements compared to the pivot in the assigned subsequence. Smaller elements need to be moved to the left and bigger elements to the right, but threads do not know where to write the element in memory. To ensure correctness, the threads need to communicate with each other and be synchronized. A mutex on a global counter can ensure that only one thread at a time will access the memory, but this approach is not very efficient for devices that use thousands of threads. It is better to calculate a prefix sum on *smaller* and *bigger* arrays. This way, each thread will know how many elements smaller than the pivot were read by threads with *lower id*, the same principle applies to elements bigger than the pivot. With prefix sum, the threads can propagate the information about how much space they will need to other threads efficiently without blocking other threads. It should be noted however, to partition a subsequence this way, auxiliary memory will be needed to move the elements. With this, the elements can then be moved in parallel, and without synchronization, from *arr* into their designated place in *aux*. The whole partitioning process is shown in Algorithm 4.2.1.

The last operation is mapping threads onto the created subsequences and sort them in parallel. At some points there will be more subsequences than threads available. At that stage, the threads will have to sort their assigned subsequence sequentially.

Algorithm 4.2.1: Parallel Quicksort

Input : array $A[1 \dots n]$, array $aux[1 \dots n]$

shared $pivot \leftarrow$ select a pivot from $A[1 \dots n]$
split $A[1 \dots n]$ between p threads
for *thread* $i \leftarrow 0$ **to** p **do in parallel**
    $smaller[i] \leftarrow$ compute number of elements smaller than $pivot$
    $bigger[i] \leftarrow$ compute number of elements bigger than $pivot$
$scanSmaller[1 \dots p] \leftarrow$ inclusive prefix sum of $smaller[1 \dots p]$
$scanBigger[1 \dots p] \leftarrow$ inclusive prefix sum of $bigger[1 \dots p]$
for *thread* $i \leftarrow 0$ **to** p **do in parallel**
    $startSmaller_i \leftarrow scanSmaller[i] - smaller[i]$
    $startBigger_i \leftarrow scanBigger[i] - bigger[i]$
    $aux[startSmaller_i \dots scanSmaller[i]] \leftarrow$ elements smaller than $pivot$
    $aux[startBigger_i \dots scanBigger[i]] \leftarrow$ elements bigger than $pivot$
$pivotBegin \leftarrow scanSmaller[p]$
$pivotEnd \leftarrow n - scanBigger[p]$
for $i \leftarrow pivotBegin$ ***to*** $pivotEnd$ **do**
    $aux[i] \leftarrow pivot$
$A[1 \dots n] \leftarrow aux[1 \dots n]$
sort $A[1 \dots pivotBegin - 1]$ and $A[pivotEnd + 1 \dots n]$ in parallel

4.2.3 Stopping in time

For smaller inputs the overhead needed to partition a section can be bigger than the work actually done through partitioning. Hoare, in their original paper [16], suggested to switch to a simpler algorithm once the subsequence is small enough. In both source papers [3, 4] that this work is based on, the authors suggested to use Bitonic sort. As such, we also decided to use Bitonic sort to help Quicksort to process small subsequences.

4.3 Implementation of Parallel Quicksort with CUDA

To call Quicksort, all that is needed is to include the source code and call the function `quicksort`. The function accepts an `ArrayView` as parameter and an optional lambda to be used as a comparison function. The whole function is implemented with templates to allow sorting of arbitrary data type. For types that do not have implicit `operator<` defined, a comparator needs to be passed as well. The lambda comparator, as described in section 3.4, allows the programmer to not only compare any data type, but also sort in descending

order if the lambda uses `operator>`. The lambda has to be able to compare from GPU, as such `__device__` specifier will be needed.

Quicksort internally is implemented as out-of-place sort procedure, because of this, during the initializing stage, an auxiliary array is allocated and this array has the same size as input. For bigger sequences, auxiliary array allocation may fail due to memory limits, this all depends on the memory size of the GPU. With the auxiliary array, other work variables and arrays are also allocated, but their size is constant. The overall extra space needed is $O(n)$.

Quicksort, implemented with CUDA, is composed of two phases. When the sequences are too big for one CUDA block to partition, multiple CUDA blocks will work together to partition the sequence and reduce the problem. This is labeled as *first phase* of Quicksort. The first phase is repeatedly called to partition the input until each section is small enough to be sorted independently by a single block. Afterwards, the *second phase* is called to sort each unsorted section using one CUDA block only. During the second phase, the CUDA block sorts the subsequences until they are completely sorted. The block also works completely independently and no communication with other CUDA blocks will be needed. The second phase of Quicksort is not executed immediately when a small subsequence is created. Instead, these tasks are accumulated over time and performed at once in parallel once there are no more big subsequences.

4.3.1 Host Side

Algorithm 4.3.1: Quicksort kernel launch

Input : array $Arr[start \dots end]$

```
1  auxiliary ← new array
2  cuda_tasks, cuda_newTasks, cuda_2ndPhaseTasks ← new TASK
     array
3  cuda_tasks ← (start, end)
4  while cuda_tasks is not empty do
5  |   gridSize ← initialize(cuda_tasks)
6  |   cuda_newTasks ← firstPhase<gridSize>(Arr, auxiliary,
   |     cuda_tasks)
7  |   remove small tasks from cuda_newTasks and insert them into
   |     cuda_2ndPhaseTasks
8  |   Arr ← auxiliary
9  |   cuda_tasks ← cuda_newTasks
10 secondPhase< |cuda_2ndPhaseTasks| >(Arr, auxiliary,
     cuda_2ndPhaseTasks)
```

As mentioned, Quicksort first partitions the input and then recursively applies itself on the newly created subsequences. However, this recursive approach can not be easily implemented with the CUDA API. Instead, an iterative approach will be implemented (Algorithm 4.3.1). In the first iteration, there is only one big sequence that needs to be partitioned. After partitioning, the big sequence is broken up into three parts and two of the parts need to be partitioned again. Here, the CUDA blocks will be synchronized explicitly through the host. The host will have at this point two subsequences and will map CUDA blocks to these subsequences and partition both of them at the same time. Once partitioning is done, there will be at most four subsequences. The four tasks are then again mapped to CUDA blocks and then partitioned again. The important bit of this iterative approach is that CUDA blocks have to wait for everyone else to finish before the next partitioning can be performed again. After i-th iteration of the first phase of Quicksort, there will be at most 2^i subsequences.

To be able to partition all sections at once in parallel, an array of tasks `cuda_tasks` is allocated. Each element from `cuda_tasks` holds information about the start and end of the section that needs to be partitioned. The first phase is repeatedly called until `cuda_tasks` is empty. In the implementation, `cuda_tasks` has a fixed size and with a bad pivot choice, it can quickly fill up. Once `cuda_tasks` is full, the second phase is called and one CUDA block is mapped to a leftover task that is present.

To call the first phase of Quicksort, each task first needs to be initialized. The initialization consists of choosing a pivot for the section and calculating how many CUDA blocks a section will need for partitioning and mapping them to the correct task. The total CUDA blocks needed by all sections are summed up and returned as $gridSize$.

Then, parallel partitioning is called. The partitioning is done in two steps. The first step moves all elements smaller than the pivot to the left and all elements greater than the pivot to the right. The second step consists of inserting pivots into the correct position and then creating two new tasks — one task for the left part with smaller elements, one task for the right part with bigger elements. The newly created tasks can be either inserted into `cuda_2ndPhaseTasks` or `cuda_newTasks`. The first situation occurs when the new sequence is small enough to be sorted by one CUDA block. Otherwise the subsequence is still too big and needs to be partitioned again using first phase of Quicksort and will be inserted into `cuda_newTasks`.

Once there are no more big subsequences, all tasks in `cuda_2ndPhaseTasks` are sorted in parallel using second phase of Quicksort.

4.3.2 Pivot choice

A good pivot choice is essential for the partition procedure and all the complications that can happen with a bad pivot choice are mentioned in subsec-

tion 4.1.2. In the implementation of TNL Quicksort, the pivot is chosen as the median of three elements: first, middle and last element of the section.

Another thing that should be noted is that the partitioning of Quicksort first phase is done in parallel and the absolute order of elements is not deterministic. This leads to a degree of randomization of the subsequences and can help with the pivot choice.

To lower memory usage, only the index of the pivot is used to store the value. To load the pivot from memory, two global reads will be needed. One read to pick up the pivot position and another read to load the element from global memory. This is an optimization that reduces overall memory usage of the sorting procedure.

4.3.3 First phase

The first phase of Quicksort consists of three parts. In the first part, CUDA blocks work together to move elements from the input array into the auxiliary array in such a way that all elements smaller than the pivot are on the left side and the right part will have elements greater than the pivot. Then synchronization will be performed. In the second part, pivot(s) will be inserted into the middle part. The last part of first phase is creating new tasks for the left and right subsequences so they can be sorted in the next iteration. In the implementation, the second part and the third part are implemented in the same kernel.

4.3.4 Multi block partitioning

During partitioning, the whole block works together and moves smaller elements to the left and all bigger elements to the right. Each thread in a block reads part of the sequence and calculates how many elements are greater than the pivot and how many are smaller (Algorithm 4.3.2, lines 4–9). The reading is done through coalesced memory load.

After reading is done, each thread will hold in private registers two counters — $lt_{threadIdx}$ and $gt_{threadIdx}$. This information now needs to be propagated to other threads in the block to arrange where each element will be copied into. This is done through block-wide parallel inclusive prefix sum. For thread $0 \leq j <$ `blockDim.x`, the prefix sum function returns sum of all values passed by threads $0 \ldots j$. The scan operation is called twice, once for elements smaller than pivot, once for elements greater than pivot (Algorithm 4.3.2, lines 10 and 11). Threads inside the block now know what offset to use in order not to overwrite other threads' writes.

To start moving the elements, the blocks will reserve a part of the auxiliary memory. It should be noted that the last thread in the block holds the sum of all counters, which means this thread knows exactly how many elements are smaller than the pivot and how many elements are greater than the pivot.

Algorithm 4.3.2: Parallel partitioning in the first phase of Quicksort [3]

Input : array $arr[start \dots end]$, array $aux[start \dots end]$, TASK *task*

```
1  (start_blockIdx, end_blockIdx) ← subsection managed by blockIdx
2  array reference src ← arr[start_blockIdx ... end_blockIdx]
3  shared pivot ← arr[task.pivotIndex]
4  lt_threadIdx, gt_threadIdx ← 0, 0
5  for i ← threadIdx; i < |src|; i ← i + blockDim do
6      if src[i] < pivot then
7          lt_threadIdx ← lt_threadIdx + 1
8      else if src[i] > pivot then
9          gt_threadIdx ← gt_threadIdx + 1
10 ltSum_threadIdx ← Σ_{i=0}^{threadIdx} lt_i
11 gtSum_threadIdx ← Σ_{i=0}^{threadIdx} gt_i
12 shared ltStart, gtStart
13 if threadIdx = blockDim - 1 then
14     ltStart ← atomicAdd(task.dstBegin, ltSum_threadIdx)
15     gtStart ← atomicSub(task.dstEnd, gtSum_threadIdx) −
           gtSum_threadIdx
16 ltFrom ← ltSum_threadIdx − lt_threadIdx
17 gtFrom ← gtSum_threadIdx − gt_threadIdx
18 shared array sharedSmaller ← array[0 ... ltSum_blockDim−1]
19 shared array sharedBigger ← array[0 ... gtSum_blockDim−1]
20 for i ← threadIdx; i < |src|; i ← i + blockDim do
21     if src[i] < pivot then
22         sharedSmaller[ltFrom] ← src[i]
23         ltFrom ← ltFrom +1
24     else if src[i] > pivot then
25         sharedBigger[gtFrom] ← src[i]
26         gtFrom ← gtFrom +1
27 for i ← threadIdx; i < ltSum_blockDim−1; i ← i + blockDim do
28     aux[ltFrom+i] ← sharedSmaller[i]
29 for i ← threadIdx; i < gtSum_blockDim−1; i ← i + blockDim do
30     aux[gtFrom+i] ← sharedBigger[i]
31 for i ← task.dstBegin to task.dstEnd do in parallel
32     aux[i] ← pivot
```

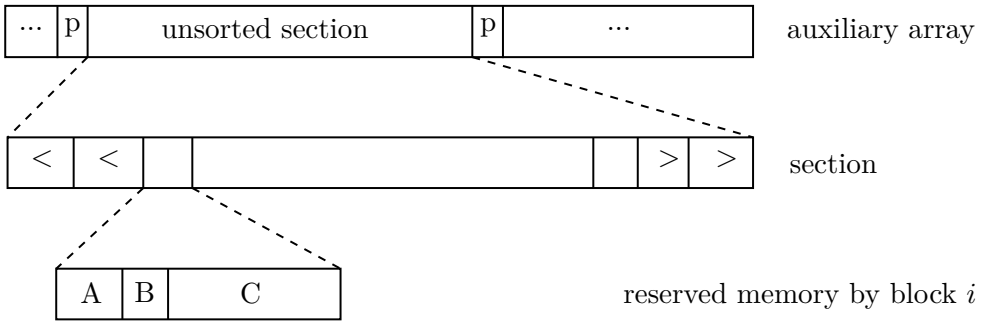

Figure 4.3: Distribution of auxiliary array during Quicksort:
(P) pivots inserted from previous iterations of Quicksort,
(A) reserved memory space for threads with $id < j$ in CUDA block i,
(B) reserved memory space for thread j in CUDA block i,
(C) reserved memory space for threads with $id > j$ in CUDA block i.

This value will be used to reserve space by moving the `task.dstBegin` and `task.dstEnd` counters (Algorithm 4.3.2, lines 12–15). These two values are in global memory space and are shared with other CUDA blocks too, for this reason, reading and modification have to be done atomically. This is achieved by calling `atomicAdd`. It is guaranteed that no other CUDA block will access the reserved part of the auxiliary memory.

The *starts* of memory for smaller and bigger elements are then written into shared memory so that other threads also know which part of the auxiliary array can be used. Because shared memory was modified, block synchronization has to be performed. The two values are labeled as *ltStart* and *gtStart*.

Each thread now knows which part of the auxiliary memory the CUDA block can modify and what the relative order of the elements will be with the help of parallel prefix sum. The last operation that needs to be done is moving the elements from the input array into the auxiliary array. The full global view on how memory is split is shown in Figure 4.3.

4.3.5 Moving elements

The whole CUDA block has reserved part of the auxiliary array with the help of `atomicAdd`, but this subsection still needs to be divided between the threads inside the block. Each CUDA thread can calculate the first unused memory address for the thread by calculating *exclusive* prefix sum of lt_i and gt_i (Algorithm 4.3.2, lines 16–17). The exclusive prefix sum holds the sum of all counters by threads with *lower id*.

To achieve both coalesced read and write, in [4], the author suggested copying the elements into shared memory first and then copy the elements from shared memory into auxiliary array (Algorithm 4.3.2, lines 18–30). The first part of copying starts with the thread reading the original element and if the el-

ement is smaller than the pivot, it gets copied into *sharedSmaller*[*ltFrom*] (Algorithm 4.3.2, lines 20–23). The next smaller element read by the thread needs to be copied into *aux*[*ltFrom + 1*] and so on. Similarly, for elements bigger than the pivot, *sharedBigger*[*gtFrom*] is used. It is sufficient for $|sharedSmaller| + |sharedBigger| \leq |src|$. After shared memory has been filled, the threads need to be synchronized and then they copy in parallel from shared memory into global auxiliary memory (Algorithm 4.3.2, lines 27–30).

With shared memory used as a buffer, coalesced read is achieved and the store operation happens very fast as shared memory is on chip. Coalesced write into global memory is also achieved as the neighbouring elements in shared memory will also be neighbours in the auxiliary array. Using this, the full bandwidth of the GPU can be used to move the elements.

4.3.6 Writing pivot

After partitioning, all elements smaller than the pivot have been moved to the left and all elements bigger than the pivot have been moved to the right. Only in the middle part, the pivot(s) have not been written in the correct position. Because the CUDA Programming model restricts synchronization between CUDA blocks, this has to be handled in a new kernel launch to assure that all CUDA blocks have moved their elements into the auxiliary array and the `task.dstBegin` and `task.dstEnd` variables have been updated correctly. These two values are key to calculating where the pivot will be inserted. The correct position is between the counters `dstBegin` and `dstEnd` that have been moved around during the partitioning. The writing of pivots is done in parallel by only *one* CUDA block (Algorithm 4.3.2, lines 31–32).

4.3.7 Creating new tasks

With the pivot inserted, the whole section is now partitioned with the left part holding all elements smaller than the pivot, the middle part has all elements equal to the pivot, and in the right part are elements greater than the pivot. The next step of Quicksort is to sort the left part and the right part. These two sections need to be created and added into the queue of works and then partitioned in the next iteration. This is handled in the same kernel as the pivot writing kernel after the pivots have been inserted.

The insertion can be done with only one thread and other threads can retire. The insertion starts with the one active thread reserving memory for one task. This is done through using `atomicAdd`, as other CUDA blocks, which working on other partitions, might want to insert new tasks too. The new task can be either inserted into `cuda_2ndPhaseTasks` or `cuda_newTasks`, depending on the size of the task.

With new tasks created, an iteration of parallel partitioning is done.

4.3.8 Second phase

In the second phase, each CUDA block works independently on a task without synchronizing with other blocks. Because of this, a simpler version of parallel Quicksort is implemented. The idea for partitioning is the same as in the first phase, but the newly created tasks need to be handled by the same CUDA block. For parts smaller than a threshold (*bitonicSize*), the sorting procedure switches from Quicksort to Bitonic sort. Bitonic sort works well for smaller inputs and the overhead is much smaller compared to Quicksort.

Algorithm 4.3.3: Second phase of parallel Quicksort

Input : array *arr*, array *aux*

```
1  shared workStack ← ∅
2  workStack ← (0, |arr|)
3  while workStack is not empty do
4      shared (begin, end) ← pop from workStack
5      if end − begin ≤ bitonicSize then
6          BitonicSort( arr[ begin... end ] )
7      else
8          shared pivot ← pick pivot from arr[ begin... end ]
9          pivotBegin, pivotEnd ← use pivot to partition arr[
             begin... end ] into aux[ begin... end ]
10         arr[ begin... end ] ← aux[ begin... end ]
11         push bigger task of (begin, pivotBegin) and (pivotEnd, end)
             into workStack
12         push smaller task of (begin, pivotBegin) and (pivotEnd, end)
             into workStack
```

4.3.9 Single block Quicksort

As mentioned, Quicksort comprises of parallel partitioning and then recursive application on the left part and the right part. The partitioning procedure works similarly as described in subsection 4.3.4. First, a pivot is chosen — the pivot is calculated as the median of three elements. Then, smaller and bigger elements are counted. Afterwards, parallel inclusive prefix sum is called. All the steps so far are the same as in the first phase. The difference now comes with how memory in the auxiliary array is reserved. Because only one block is partitioning the section, `atomicAdd` is not needed to communicate with other CUDA blocks and only shared variables are used to propagate information about the reserved section to other threads. Elements are then copied from source to auxiliary through shared memory as a buffer to achieve

both coalesced read and write — the method was described in subsection 4.3.5. The last operation of partitioning is writing the pivot into the correct position.

To finish sorting, the left part (`begin`–`pivotBegin`) and the right part (`pivotEnd`–`end`) need to be sorted too. To sort these sections, the implementation uses a stack to substitute stack frames needed for recursion.

4.3.10 Explicit stack

Each kernel has a limited number of fast local registers and upon using them up, spilled memory is stored into global memory. To minimize the usage of the implicit stack frame, the original paper [3] suggested implementing a stack in shared memory to substitute recursion. The stack only saves *begin* and *end* positions of the subsequence that still needs to be sorted. The stack is a static array allocated in shared memory and the size of shared memory is known at compile time. By default, the shared memory is allocated to be able to handle recursion of depth 32. Worth noting is the similarity between a stack frame used in the second phase and a CUDA task used in the first phase (an element of `cuda_tasks`). Both serve the same purpose — to indicate which part of the sequence has not been sorted yet.

The start of the function starts with picking up a section that needs to be partitioned (Algorithm 4.3.3, line 4). Only one thread is needed to manipulate with the stack and all necessary information is saved in shared memory to broadcast the information to other threads. Synchronization here is necessary to guarantee the update of values.

Small subsequences are sorted using Bitonic sort, but bigger subsequences need to be partitioned first (Algorithm 4.3.3, lines 7–10). After partitioning, two new tasks are created and they need to be pushed into the stack (Algorithm 4.3.3, lines 11-12). Only one thread is needed to execute this operation. To guarantee $O(\log n)$ recursion depth, in [21], the author proposed to insert the bigger part first and then the smaller part. This way, the smaller part is sorted first and stack frame is immediately freed.

4.4 Optimizations

To gain an optimized implementation of Quicksort, some changes need to be made. The first optimization was already described in subsection 4.3.5, where shared memory was used to achieve coalesced reads and writes. Other changes will be described in the following chapters.

4.4.1 Parallel prefix sum

Prefix sum is an essential operation used in partitioning operation to speed up the arrangement of elements. To gain an even more efficient algorithm,

prefix sum was implemented (Algorithm 4.4.1) to run in parallel and reduce the number of operations needed.

The parallel prefix sum, as implemented, only needs local registers of threads and a static shared array of size 32. The function requires *every* thread in the block to contribute a value into the scan operation and for thread i, the function returns the sum of all values given by threads $0 \dots i$.

First, parallel scan is performed on a warp level. Warps are implicitly synchronized on the hardware level due to how they are executed. Scan is here used with the help of `shuffle` intrinsic. Data is passed between threads in a warp through registers and no shared memory is needed for this stage.

Let us observe what value each thread has after the warp scan was performed. For warp group 0 (threads $0 \dots 31$) all threads have the correct value. For warp group 1, thread $32 \leq p < 64$, only has values from threads $32 \dots p$.

$$warpScan_p = \sum_{i=32}^{p} x_i$$

To calculate the the correct values, every thread from warp group 1 still needs to add all the values from warp group 0. Let us also notice that warp 31 holds the sum of all values from warp group 0. This value can be shared between threads in warp group 1 to finish the calculation. For warp group 2 (threads $64 \dots 95$), every thread needs sum of both warp group 0 and 1 (threads $0 \dots 63$). This missing value can be taken from $warpScan_{31}$ and $warpScan_{63}$. Similarly, for the rest of the warp groups, each thread needs the values $warpScan_{32i-1}$.

With this bit of information, the last threads from each warp group need to share their value with other threads in the CUDA block. This is achieved by using a shared $array[0 \dots 31]$ and synchronizing all threads after the shared array has been filled. The maximum CUDA block size is 1024 and warp size is 32, because of this, at most $32 = \frac{1024}{32}$ variables are needed.

The last bit of parallel scan optimization is done through transforming the shared array into exclusive scan of the array itself, i.e. $array'_i = \sum_{j=0}^{i-1} array_i$. This optimization is done so that thread 1023 will not have to access $array$ 31 times to add up the missing values.

The whole procedure is written in pseudo code again in Algorithm 4.4.1.

4.4.2 Optimization with array rotation

Quicksort as implemented, uses an auxiliary array for partitioning. To continue the next iteration of partitioning, the input would need to be copied from auxiliary array back to input array. For smaller inputs, the time needed to copy could be negligible, but for big arrays, each copy could pose a serious bottleneck to the function. For this reason, an approach of *source rotation* is chosen.

Algorithm 4.4.1: CUDA parallel inclusive prefix sum

Input : *value*

1 $warpId, warpGroup \leftarrow$ calculate warp identification values
2 shared $array[0 \ldots 31]$
3 $warpScan_{threadId} \leftarrow$ warpInclusiveScan($value$)
4 **if** $warpId = 31$ **then**
5 $\quad array[warpGroup] \leftarrow warpScan_{threadId}$
6 **if** $warpGroup = 0$ **then**
7 $\quad array[warpId] \leftarrow$ warpExclusiveScan($array[warpId]$)
8 **return** $warpScan_{threadId} + array[warpGroup]$

In the first iteration, the data is in the input array and the partitioning function moves the elements into the auxiliary array. In the next iteration, the threads will read data from the auxiliary array and move them back into the original array. This solution saves time by not copying the arrays after every iteration, but the overhead is now higher due to the need to check which array actually holds the data.

To keep track of the current source array, a simple counter is used — `iteration`. The counter denotes how many iterations of partitioning have been performed. For even `iteration`, the data can be found in the input array, whereas for odd `iteration`, the data that needs to be partitioned is in the auxiliary array.

The same optimization can be used for `cuda_tasks` and `cuda_newTasks` (Algorithm 4.1.1). When `iteration` is even, `cuda_tasks` array will hold the current tasks and new tasks will be written into `cuda_newTasks`. For odd `iteration`, the two arrays are interchanged.

Care should be given to the sorting procedure so that the sorted results are correctly inserted back into the input array. A problem arises for elements in resting position, i.e. the elements will not be moved again. The only case when this can happen is when pivots are inserted or when the small subsection is sorted using Bitonic sort. For pivots, the solution is to always write into the input array. The pivot writing procedure is done by only one CUDA block and it is guaranteed that no thread will be reading from global memory at the same address.

With small sequence being sorted by Bitonic sort, the solution is also not complicated. In the implementation, Bitonic sort is used if the subsequence can be copied into shared memory. Once the subsequence is sorted in shared memory, block synchronization is performed and then the result is copied directly into the input sequence.

4.4.3 Elements per CUDA block

Special care was also given to the distribution of tasks during the first phase of Quicksort. For a subsequence of length n and CUDA blocks with 512 threads, it is necessary to decide how many blocks will be needed to partition the subsequence. It is possible to map each thread to one element. Using this, $\lceil \frac{n}{512} \rceil$ blocks will be needed. The other extreme is using only one CUDA block and map $\lceil \frac{n}{512} \rceil$ elements per thread and thus most of parallelism will be lost.

In the implementation, each thread processes at most 8 elements. With this, the number of CUDA blocks will be lowered and the overall number of `atomicAdd` calls is also lowered, while still having enough CUDA blocks to occupy the hardware. The number 8 was chosen after careful tuning and profiling to get a good overall result. The number of elements per thread is further adjusted during run-time to allow use of shared memory for the moving part of the first phase of Quicksort (subsection 4.3.5). With too many elements being handled by a CUDA block, there is a chance that the whole part will not fit into shared memory (`elemPerBlock*sizeof(arr[0]) <= sharedMemSize`).

4.5 Using CUDA dynamic parallelism

With the help of CDP, it is possible to remove the iterative part of first phase of Quicksort and launch newly created tasks directly inside the kernel. In this section, two approaches will be described on how to leverage nested kernel launch to sort independently. This section is mainly of theoretical character.

4.5.1 Version 1

The first version of CDP Quicksort uses CDP to sort new tasks. In the original implementation of Quicksort first phase, after the first iteration, the whole input sequence was partitioned into three parts. The left part held elements smaller than the pivot, the middle part had elements equal to the pivot and the right part consisted of elements greater than the selected pivot. Then synchronization with CPU had to be done and only afterwards could the left part and right part be partitioned in parallel again. With CDP, we propose to use multiple CUDA blocks to partition the input and then only the last active CUDA block will launch new CUDA kernels to process left and right parts (Algorithm 4.5.1 lines 6–20). The important part is how to recognize which CUDA block is the last one working. To solve this, *task* will hold a counter labeled as `stillWorking`. This counter was initialized with the number of blocks allocated to worked together to partition the subsequences ($lBlocks$ and $rBlocks$). Each time a CUDA block is done with moving all its elements, one `atomicAdd(&task.stillWorking, -1)` will called. The last working CUDA block will get 1 as the return value.

Algorithm 4.5.1: Quicksort with CDP version 1

Input : array $arr[1 \dots n]$, array $aux[1 \dots n]$, TASK $task$

```
1  if gridDim = 1 then
2      deallocate task
3      QuickSort_2ndPhase(arr[1...n], aux[1...n])
4  else
5      isLast ← partition(arr[1...n], aux[1...n], task)
6      if isLast = False then
7          return
8      shared (pivotBegin, pivotEnd, pivot) ← task
9      i ← pivotBegin + threadIdx
10     for ; i < pivotEnd ; i ← i + blockDim do
11         aux[i] ← pivot
12     arr[1...n] ← aux[1...n]
13     if threadIdx ≠ 0 then
14         return
15     deallocate task
16     if pivotBegin > 0 then
17         lBlocks ← calcBlocksNeeded(pivotBegin)
18         lpivot ← pick pivot from arr[1...pivotBegin − 1]
19         leftTask ← new (1, pivotBegin, lBlocks, lpivot)
20         QuickSort<lBlocks, new stream>(arr[1...pivotBegin − 1],
             aux[1...pivotBegin − 1], leftTask)
21     if n − pivotEnd > 0 then
22         rBlocks ← calcBlocksNeeded(n − pivotEnd)
23         rpivot ← pick pivot from arr[pivotEnd + 1...n]
24         rightTask ← new (pivotEnd + 1, end, rBlocks, rpivot)
25         QuickSort<rBlocks, new stream>(arr[pivotEnd + 1...n],
             aux[pivotEnd + 1...n], rightTask)
```

The last CUDA block can safely insert pivots into global memory (Algorithm 4.5.1 lines 6–11). Once pivots are inserted, the new tasks need to be sorted in parallel. To start new kernels, only 1 CUDA thread will be needed and all other threads can retire (Algorithm 4.5.1 lines 13–14). Then thread 0 creates new kernels that use one or more CUDA blocks to sort a subsection. Worth noting here is that to enable concurrent sorting, each kernel launch has to be in its own CUDA Stream. Failing to sort them in different streams leads to serialization of the kernel launches, i.e. left part will be sorted completely first and only then will the recursive sort on the right part be executed.

Another interesting part is how to enable communication between blocks with the use of `atomicAdd` during the partitioning part. The *task* has to be allocated in global memory so that every CUDA block can access it. Then, a problem arises with which CUDA block will have the responsibility to deallocate the created *task*. The solution chosen in this work is to wait until the last CUDA block is completely done with partitioning and only then free up the memory (Algorithm 4.5.1 lines 2–15).

4.5.2 Version 2

Algorithm 4.5.2: Quicksort with CDP version 2

Input : array $arr[1 \dots n]$, array $arr[1 \dots n]$

1 $pivot \leftarrow$ pick pivot from $arr[1 \dots n]$
2 $blocksNeeded \leftarrow$ calculate blocks needed to partition $arr[1 \dots n]$
3 $task \leftarrow$ new $(1, n, pivot, blocksNeeded)$
4 partition<*blocksNeeded*, new *stream*>($arr[1 \dots n]$, $arr[1 \dots n]$, *task*)
5 $arr[1 \dots n] \leftarrow arr[1 \dots n]$
6 (*pivotBegin, pivotEnd, pivot*) $\leftarrow$ *task*
7 deallocate task
8 **if** $pivotBegin - 1 > 0$ **then**
9 QuickSort<new *stream*>($arr[1 \dots pivotBegin - 1]$, $arr[1 \dots pivotBegin - 1]$)
10 **if** $end - pivotEnd > 0$ **then**
11 QuickSort<new *stream*>($arr[pivotEnd + 1 \dots end]$, $arr[pivotEnd + 1 \dots end]$)

Definition 4.1. A controller CUDA thread is a thread that can launch new kernels. A worker CUDA thread is a thread that executes a cannot launch new kernels.

To start CDP Quicksort version 2, the CPU only needs to launch the kernel with one CUDA thread. The one thread will perform the role of a controller

and control the flow of Quicksort. First, the input needs to be partitioned. In this part, the controller thread will start a new kernel of workers to partition the input (Algorithm 4.5.2, lines 1–6). The partitioning can be performed the same way as in the regular first phase of Quicksort. After the partition kernel has been launched, it is necessary to synchronize the workers and the controller. Once partitioning is done, the controller will know how big the left part and the right part is. The controller can then launch two new kernels. One controller to sort the left part and another to sort the right one. Same as with version 1, the newly created controller kernels need to be in separate streams.

In total, version 2 of CDP Quicksort uses parallelism in two ways. One kernel launch is used for partitioning and another two kernel launches are used in order to sort in parallel.

Chapter 5

Testing and measuring

This chapter will be about testing and measuring the time needed for execution of the implemented algorithms.

5.1 Environment

All measurements were done on `gp1` systems hosted by Faculty of Nuclear Sciences and Physical Engineering, CTU in Prague. The specifications are as follows:

	gp1
CPU	Intel Xeon CPU E5-2630 v3 @ 2.40GHz
GPU	NVIDIA Quadro P6000
RAM	125 GB
OS	Arch Linux 5.10.5-arch1-1
nvcc	11.2
g++	(GCC)10.2.0
driver	460.323.0

Table 5.1: Environments used to perform measuring and testing.

On the `gp1` machine, a professional Quadro GPU with 24 GB was used.

To compile the source code, `nvcc` is used in combination with local `g++` compiler to link libraries. The compilation flags are `-O3` with `-std=c++14` and debugging from the TNL library has been turned off with `-DNDEBUG`.

5.2 Testing

For each implemented algorithm, a set of unit tests is present. The tests ensure that the algorithms run correctly and can be incorporated into the TNL

```
template <typename Value, typename Function>
bool is_sorted(ArrayView<Value, Devices::Cuda> arr
                const Function &Cmp)
{
    if (arr.getSize() <= 1)
        return true;

    auto fetch = [=] __cuda_callable__(int i)
                    { return !Cmp(arr[i], arr[i - 1]); };
    auto reduction = [] __cuda_callable__(bool a, bool b)
                    { return a && b; };
    return TNL::Algorithms::Reduction<Devices::Cuda>::reduce
            (1, arr.getSize(), fetch, reduction, true);
}
```

Listing 6: Implementation of is_sorted.

library. The tests have been performed multiple times and all tests passed on every launch. It should be noted that the algorithms run in parallel and the execution order of threads is not deterministic. Therefore, it is impossible to guarantee complete correctness of the implementations, but with the extent of testing done, no bugs have been found yet.

The tests not only assure that the sorting is done correctly but also make sure that the templating is implemented correctly. The unit tests contain sorting of non `integer` values, sorting with lambdas, sorting structures.

To speed up the checking process, a simple `is_sorted` was implemented. The function (Listing 6) uses parallel reduction provided by the TNL library to check if every two consecutive elements are in the correct order. The function uses `operator <` and as such, the actual check is done by testing whether element $a_{i+1} < a_i$ (this should fail for sorted input).

All tests were created with the help of `gTest` [22] framework provided by Google.

5.3 Methods of measuring

To give a better understanding of the performance of the implementations, the functions were measured on the `gp1` machine. For sorting purposes, it is assumed that all data is already on GPU and only the time of sort without copying from CPU and GPU was measured.

To measure the time elapsed between the function call and when the control is returned, a `TIMER` class was implemented. The object measures the time elapsed between creation of the object and when the destructor is called. The constructor of the class takes a lambda as a parameter and

```
1 Array<Value, Devices::Cuda> arr(data);
2 auto view = arr.getView();
3 {
4     TIMER t([](double res){std::cout<<res<<std::endl;});
5     quicksort(view);
6 }
```

Listing 7: Example of `TIMER` usage, block scope is used to forcibly call destructor. The timer prints on standard output the measured time.

it is used as a call-back to evaluate, what to do with the result, once the destructor is called. The default call-back prints the time on the standard output. The result is of type double and represents how many milliseconds have been measured. The time is measured as difference between each call of `std::chrono::high_resolution_clock`. An example on how to use the timer is in Listing 7.

All measurements were done 20 times and then the final result was calculated as the average of all measurements.

Worth noting is that CUDA kernel launches are non-blocking, therefore it is necessary to at least call `cudaDeviceSynchronize()` to ensure that all kernels have finished.

5.3.1 Testing data sets

For input data, different kinds of sequences were generated. The sequence types are:

Random A sequence with random values, each value has the same probability of being generated. The data was generated with `std::rand()` after being seeded.

Shuffle `std::random_shuffle` was used to permutate the sequence $1 \dots n$.

Sorted A sorted sequence $0 \dots n-1$.

Almost sorted A sorted sequence $0 \dots n-1$ but three random swaps were done on the array.

Decreasing Sequence $n \dots 1$.

Zero entropy All values are the same.

Gaussian distribution Each value is created by calculating the arithmetic average of four randomly picked values from uniform distribution.

Bucket The data set is divided into p blocks and each block is further divided into p sections, where $p > 0$. Section $i > 1$ contains randomly selected values between $(i-1)\frac{2^{31}}{p}$ and $i\frac{2^{31}}{p} - 1$.

Staggered The data set is divided into p blocks. The staggered distribution is then created by assigning values for block i, where $i \leq \lfloor \frac{p}{2} \rfloor$, so that they all lie between $(2i+1)\frac{2^{31}}{p}$ and $(2i+2)\frac{2^{31}}{p} - 1$. For blocks where $i > \lfloor \frac{p}{2} \rfloor$, the values lie between $(2i-p)\frac{2^{31}}{p}$ and $(2i-p+1)\frac{2^{31}}{p}$.

The last three distributions (Gaussian, Bucket, Stagger) were originally proposed and implemented in [3].

5.3.2 Comparison with other implementations

For Bitonic sort, there were not many efficient implementations using CUDA. Most of them were either very basic or did not work properly. However, NVIDIA sample implementations [2] contain an efficient implementation and it was used as a baseline for measurement.

With Quicksort, the parallel implementations that were chosen are as follows:

1. cdpAdvancedQuickSort provided by NVIDIA sample library [2],
2. GPUQuicksort by Cederman et al. [3],
3. CUDAQuickSort by Manca et al. [4].

To compare with other algorithms too, `std::sort` was chosen as baseline for sequential sort. As for other parallel algorithms, `thrust::sort` [5] from the CUDA toolkit was selected. The sort provided by the thrust library is based on Radix sort and can not be directly compared to Quicksort. This is due to how the algorithms work. Quicksort is a comparison based algorithm that uses `operator<` to split up the input, while Radix sort uses base decomposition of the numbers to split the input into buckets in order to sort the input [23]. Despite this, the results are included to provide a better context for how much faster parallel sort can be, compared to a single threaded implementation.

One of the requirements for this work was to compare the implementation with [24]. This was not done due to the complicated structure of the source code. With all the dependencies throughout the whole code base, it was impossible to include or refactor out the needed functions to create a runnable code. Furthermore, the implementation provided by [24] uses `<Windows.h>` header file, which makes it also impossible to compile and execute in a Linux environment. Because of all the previously mentioned reasons, the implementation provided by the supervisor was not used.

5.3.3 Results

All mentioned algorithms were measured using the distributions mentioned in subsection 5.3.1. As baseline, `std::sort` was used and for other algorithms, the number indicates what the *speed-up* is compared to `std::sort`.

size	STL t[ms]	NV_{Bt} S	Bit S	ced_{Qs} S	man_{Qs} S	NV_{Qs} S	Qs S	$thrust$ S
10	0.033	1.65	0.22	0.00	0.03	0.00	0.07	0.94
11	0.074	3.21	0.48	0.03	0.08	0.01	0.15	2.31
12	0.172	5.73	1.06	0.07	0.18	0.04	0.31	5.21
13	0.377	9.19	2.17	0.15	0.38	0.09	0.46	2.99
14	0.780	13.92	4.17	0.33	0.77	0.19	0.84	6.04
15	1.685	18.11	7.23	0.76	1.59	0.40	1.56	12.86
16	3.540	23.91	11.91	1.55	3.16	0.90	2.37	23.13
17	7.612	30.69	17.82	3.07	6.47	1.83	4.59	40.70
18	15.924	33.45	24.84	5.19	11.28	2.93	8.43	43.50
19	33.741	20.20	28.86	8.26	19.13	5.31	14.12	64.39
20	71.678	20.47	27.93	8.39	31.77	8.42	22.64	109.26
21	146.668	19.08	27.10	7.07	47.25	11.66	31.38	161.52
22	306.145	17.75	26.45	6.79	57.69	14.21	40.86	225.43
23	647.072	16.62	25.79	6.85	68.52	17.57	47.28	280.11
24	1335.382	15.18	24.49	6.80	72.88	18.94	51.35	310.19
25	2789.663	14.10	23.50	6.65	80.00	16.15	54.35	342.87

Table 5.2: Speed-up of algorithms compared to `std::sort` for various input sizes. **Random** distribution was used as input. The first column is input size in $\log_2$ scale. The second column is time it took `std::sort` to sort the input in milliseconds on `gp1`. All other values are speed-up compared to `std::sort`. The implementations used are:
(STL) `std::sort` from the STL library [25],
(NV_{Bt}) NVIDA's implementation of Bitonic sort [2],
(Bit) TNL implementation of Bitonic sort,
(ced_{Qs}) Cederman et al.'s implementation of Quicksort [3],
(man_{Qs}) Manca et al.'s implementation of Quicksort [4],
(NV_{Qs}) NVIDA's implementation of Quicksort with CDP [2],
(Qs) TNL implementation of Quicksort,
($thrust$) thrust's implementation of Radix sort in `thrust::sort` [5].

size	*STL* t[ms]	NV_{Bt} S	*Bit* S	ced_{Qs} S	man_{Qs} S	NV_{Qs} S	*Qs* S	*thrust* S
10	0.033	1.73	0.23	0.01	0.03	0.00	0.07	1.03
11	0.075	3.26	0.49	0.03	0.08	0.01	0.15	2.34
12	0.167	5.56	1.03	0.07	0.17	0.04	0.30	5.06
13	0.363	8.85	2.09	0.15	0.36	0.08	0.49	2.40
14	0.750	13.39	4.01	0.31	0.71	0.18	0.80	4.83
15	1.544	16.78	6.74	0.66	1.19	0.36	1.35	9.89
16	3.137	22.73	10.49	1.31	2.35	0.79	2.05	17.52
17	6.441	26.07	16.06	2.50	4.80	1.52	3.68	32.04
18	13.834	29.18	21.61	4.35	9.23	2.41	6.94	36.79
19	29.384	17.64	25.15	7.09	15.12	1.80	11.75	53.52
20	60.164	17.30	23.46	6.78	23.86	0.75	18.09	87.06
21	121.900	15.95	22.54	6.33	32.95	0.71	23.70	127.64
22	250.738	14.62	21.68	6.23	40.84	0.69	26.91	176.20
23	582.956	15.04	23.23	6.21	51.44	0.74	28.14	244.52
24	1338.513	15.29	24.53	6.92	62.40	0.80	27.69	314.42
25	2943.701	14.95	24.77	7.37	70.05	0.82	22.96	350.69

Table 5.3: Speed-up of algorithms compared to `std::sort` for various input sizes. **Staggered** distribution was used as input. The first column is input size in $\log_2$ scale. The second column is time it took `std::sort` to sort the input in milliseconds on `gp1`. All other values are speed-up compared to `std::sort`. The implementations used are:
(*STL*) `std::sort` from the STL library [25],
(NV_{Bt}) NVIDA's implementation of Bitonic sort [2],
(*Bit*) TNL implementation of Bitonic sort,
(ced_{Qs}) Cederman et al.'s implementation of Quicksort [3],
(man_{Qs}) Manca et al.'s implementation of Quicksort [4],
(NV_{Qs}) NVIDA's implementation of Quicksort with CDP [2],
(*Qs*) TNL implementation of Quicksort,
(*thrust*) thrust's implementation of Radix sort in `thrust::sort` [5].

In Table 5.2, the speed-up is shown with Random distribution. To describe, the baseline implementation of `std::sort` works very fast for smaller inputs but is overtaken by other parallel algorithms for bigger sequences.

TNL Bitonic sort is faster than `std::sort` with input being bigger than 2^{12} but not much speed-up is gained with sequences bigger than 2^{18}. This is an expected result as Bitonic sort has to do many global memory operations for very big inputs and the time complexity is $O(\log^2 n)$ only. It is worthwhile to mention that Bitonic sort will always do the same number of phases no matter the distribution. As such, the time will not change significantly with

a different distribution.

Quicksort on the other hand is slower for smaller inputs due to its high overhead but quickly overtakes Bitonic sort for inputs bigger than 2^{20}. The implementation of TNL Quicksort is also consistently faster than NVIDIA's Quicksort and the Quicksort implemented by Cederman. When compared with Manca's more optimized sort, our implementation is $\approx$ 1.4x slower. A hypothesis on why our implementation is slower is given in subsection 5.4.2.

A notable observation can be made by comparing the algorithms with a staggered distribution. The complete results are in Table 5.3. The speed-up of TNL Quicksort when compared to `std::sort` is only $\approx$ 20x for bigger inputs. With the implemented method of picking pivot as median of 3 elements, this kind of distribution forces the algorithm to choose a bad pivot most of the time. With a bad pivot, the partitioning phase will not be able to split the subsequence evenly and this leads to more partitioning being needed. Manca's implementation appears to slow down a little bit but the performance penalty is not as big. The comparison shows how important a good pivot choice is to the algorithm.

Other distributions were also used to measure the sorting time and are included in the medium. The results are in line with measurements done with Random distribution.

5.4 Profiling

Both of the implemented algorithms were profiled using NVIDIA NSight Systems [26] and compared to baseline implementations. The profiling was done on the local machine with GTX 1650 laptop GPU and the implementations sorted an `integer` sequence of length 2^{25} with Random distribution. The executable binaries were started using `nsys profile` and the application emitted a `.qdrep` file. The file was then used for visualization in Nsight systems and the screenshots of the application can be found in the included medium.

5.4.1 Bitonic sort

TNL Bitonic sort was profiled and compared with CUDA Samples' [2] implementation of Bitonic sort. The main kernels that were compared are: the `first phase`, where shared memory was used to sort the beginning, then the merge stage that was done in global memory, also labeled as `global merge`, and finally `merge shared`, an operation that merges in shared memory (Algorithm 3.4.2). Both implementations used a configuration of 512 threads per CUDA block and 32765 CUDA blocks to sort the sequence. The results can be seen in Table 5.4.

The `global merge` and `shared merge` measured times were averaged as each run took a different amount of time due to the scheduler and data dependencies during swapping. In total, TNL Bitonic sort was faster in every

	CUDA Bitonic	TNL Bitonic
`first phase`	44.4 ms	36.6 ms
`global merge`	3.8 ms	1.3 ms
`shared merge`	6.5 ms	6 ms

Table 5.4: Comparison of kernels of CUDA Bitonic sort against TNL Bitonic sort for integer types.

phase of the run. This might be due to the fact that CUDA Bitonic sort uses a sorting procedure that sorts both keys and values. With the data being double the size, more memory access were needed. This is especially evident with `global merge`. More data needed to be fetched and it worsened the performance of kernel by a factor of 3.

To give a more fair comparison, TNL Bitonic sort was profiled again, this time with the same sequence but the data are of type `double`. The results are as follows:

	CUDA Bitonic (int, int)	TNL Bitonic (double)
`first phase`	44.4ms	47ms
`global merge`	3.8ms	2.3ms
`shared merge`	6.5ms	7.5ms

Table 5.5: Comparison of kernels of CUDA Bitonic sort that sorts (int, int) against TNL Bitonic sort doubles.

Here (Table 5.5), the data show a much more reasonable comparison. TNL Bitonic sort loses against CUDA Bitonic sort during the phases where shared memory was used, this might have been caused by bank conflicts during memory access in shared memory as `double` is not aligned to 4 bytes as required by shared memory bank. On the other hand, TNL Bitonic sort's `global merge` is 1 ms faster than the one done by CUDA Bitonic sort. This might be due to where the data are saved in global memory. For TNL Bitonic sort, the whole 512*2*8 memory block (`blockSize*2*sizeof(double)`) can be found in one place, whereas for CUDA Bitonic sort, the key and value are be in two different memory segments.

5.4.2 Quicksort

As baseline for parallel Quicksort, `CUDAQuickSort` implemented by Manca et al. [4] was chosen. Both Manca's Quicksort and TNL Quicksort work in two phases, the first phase partitions sequences in parallel and the second phase finishes sorting smaller subsequences. These two phases are the main bottleneck

of the procedure and were intensively profiled. The resulting measurement is shown in Table 5.6.

	Manca Quicksort	TNL Quicksort
`First phase` time	5 ms	3.7 ms
`First phase` count	17	27
`Second phase` time	30 ms	60 ms

Table 5.6: Results of profiling done on the first and second phases of Quicksort implemented by Manca et al. [4] compared to TNL Quicksort. First phase time denotes time of *each* kernel launch. First phase count denotes number of kernel launches. Second phase time denotes time needed to finish sorting using second phase of Quicksort.

The first big difference is the number of iterations of the first phase of Quicksort that have been done. Manca's implementation ran in total 17 times whereas TNL Quicksort had to partition 27 times with the first phase of Quicksort. This might be due to how the pivot is chosen in each of the algorithms. TNL Quicksort chooses pivot as median of 3 elements. Manca's implementation on the other hand chooses median as the average of minimum and maximum of the subsequence. Here, the main bottleneck of Manca's approach is that every element has to be compared multiple times to find the minimum and maximum. Furthermore, this policy of pivot choosing can not be generalized to sort non numeric data types as calculating average does not have to make sense for all data types. With a better pivot, Manca's implementation has to do the expensive first phase fewer times and the subsequences at the end divide the load more equally. For future works, it is possible to add a specialization of Quicksort in order to choose the pivot the same and compare the implementation. Such a change however requires major changes in the algorithm and requires more memory in order to store minimum and maximum for each partition.

Another thing to note is the configuration that was used in order to partition during the first phase. Manca's implementation used on average 32000 CUDA blocks with 256 threads (the implementation does not work with 512 threads). TNL Quicksort's implementation used on average 8200 CUDA blocks with each block having 512 threads. In subsection 4.4.3, the method was described. The resulting work maps at most 8 elements per CUDA thread and reduces the number of CUDA blocks. As a result, TNL Quicksort has less overhead needed to switch context for CUDA blocks but more importantly, fewer `atomicAdd` need to be called in total. This all leads to TNL Quicksort taking less time for the first phase of parallel Quicksort.

For the second phase, TNL Quicksort works twice as slow compared to Manca's Quicksort. This is mainly due to the way pivot is chosen and how

big the resulting subsequences are. Manca's approach chooses pivot as an element very close to median and spreads out the work load very evenly. Then, Bitonic sort during the second phase has very little work to do to finish its task. TNL Quicksort on the other has a limited `task queue` and switches to second phase once there are more than 2^{14} tasks. Some subsequences are still too big and are further partitioned by Quicksort in one CUDA block. This leaves a big penalty on the performance. The queue size was also expanded to exchange memory needed to store tasks for more partitioning in order not to overwhelm the second phase, but no visible improvements were made this way. All the time that was saved in the second phase was transferred to the first phase and more first phase iterations were executed.

In total, TNL Quicksort runs slower than Manca's implementation. Profilation showed that the problem lies in the way the pivot is chosen. The partitioning procedure runs faster than Manca's implementation but more partitioning needed to be done.

Conclusion

Goals and results

The goal of this work was to study and implement efficient sorting algorithms for GPUs, namely Bitonic sort and Quicksort. First, the algorithms were described in detail and then the implementations were presented. Interesting and important sections of the code were commented to explain the thought process behind the made decisions. Finally, the presented work was tested, measured, and compared with known implementations for CPU and GPU.

This thesis lays the groundwork for sorting primitives to be added into the TNL library. TNL is mainly written in C++ extended with CUDA, as such the code was also written in C++.

The resulting work contains the implementation of both previously mentioned algorithms. For Bitonic sort, a CPU version is available and a version for a CUDA block is also present. This version of Bitonic sort can sort inputs of any type and of any length. Shared memory was used in combination with other techniques to gain speed-up. Bitonic sort was tested for various inputs of different lengths and different types. Measurement against CPU `std::sort` show $\approx$ 25x speed-up for sequences of length 2^{18}–2^{25} with 32 bit integers.

With the help of block Bitonic sort, an efficient version of Quicksort was created based on Cederman et al.'s [3] and Manca et al.'s [4] work. The implementation can also sort inputs of any type and of any length. Compared to Bitonic sort, TNL Quicksort is slower for smaller inputs but quickly gains speed-up for sequences of length greater than 2^{21}. However, when compared to Manca et al.'s CUDAQuicksort or `thrust::sort`, both Bitonic sort and Quicksort are slower. Against `std::sort`, Quicksort shows improvement by a factor of ≈ 50 for sequences of length 2^{24}–2^{25}. It should also be noted that at least double the memory is needed to run Quicksort as the algorithm is implemented as out-of-place algorithm.

Future work

This thesis' implementation of Quicksort, as shown in subsection 5.3.2, is slower than Manca et al.'s [4] implementation. This stems from the way the pivot is chosen. For future work, a better choice of pivot can be studied and then implemented. With a better pivot, the task load can be spread out more evenly and speed-up can be gained this way. Another improvement can be made by extending the code to support sorting on multiple GPUs at once. This can be achieved using MPI [27].

Bibliography

1. OBERHUBER, Tomáš; KLINKOVSKÝ, Jakub; FUČÍK, Radek. *Template Numerical Library* [online]. 2021 [visited on 2021-04-22]. Available from: `https://tnl-project.org/`.
2. NVIDIA. *Cuda Samples* [comp. software]. [N.d.]. Version 11.2 [visited on 2021-02-22]. Available from: `https://docs.nvidia.com/cuda/cuda-samples/index.html`.
3. CEDERMAN, Daniel; TSIGAS, Philippas. GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors. *ACM J. Exp. Algorithmics*. 2010, vol. 14. ISSN 1084-6654. Available from DOI: `10.1145/1498698.1564500`.
4. MANCA, Emanuele; MANCONI, Andrea; ORRO, Alessandro; ARMANO, Giuliano; MILANESI, Luciano. CUDA-quicksort: an improved GPU-based implementation of quicksort. *Concurrency and Computation: Practice and Experience*. 2016, vol. 28, no. 1, pp. 21–43. Available from DOI: `https://doi.org/10.1002/cpe.3611`.
5. HOBEROCK, Jared; BELL, Nathan. *thrust* [online]. 2021 [visited on 2021-04-24]. Available from: `https://github.com/NVIDIA/thrust/`. [software].
6. NVIDIA CORPORATION. *CUDA C++ Programming Guide* [online]. 2021 [visited on 2021-04-03]. Available from: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.
7. HARRIS, Mark. *Using Shared Memory in CUDA C/C++* [online]. 2013 [visited on 2021-04-03]. Available from: `https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/`.
8. MAREŠ, Martin; VALLA, Tomáš. *Průvodce labyrintem algoritmů*. 1. electronic edition. Praha: Edice CZ.NIC, 2017. ISBN 8088168198.

9. PETERS, Tim. *Python-Dev Sorting* [online]. 2002 [visited on 2021-05-10]. Available from: `https://mail.python.org/pipermail/python-dev/2002-July/026837.html`.
10. ARKHIPOV, Dmitri I; WU, Di; LI, Keqin; REGAN, Amelia C. Sorting with gpus: A survey. *arXiv preprint arXiv:1709.02520*. 2017.
11. BATCHER, Kenneth E. Sorting Networks and Their Applications. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. Atlantic City, New Jersey: Association for Computing Machinery, 1968, pp. 307–314. AFIPS '68 (Spring). ISBN 9781450378970. Available from DOI: `10.1145/1468075.1468121`.
12. *Solving the recurrence relation T(n) = 2T(n/2) + nlog n via summation* [online]. 2019 [visited on 2021-05-05]. Available from: `https://cs.stackexchange.com/questions/115274/solving-the-recurrence-relation-tn-2tn-2-nlog-n-via-summation`.
13. PETERS, Hagen; SCHULZ-HILDEBRANDT, Ole; LUTTENBERGER, Norbert. Fast In-Place Sorting with CUDA Based on Bitonic Sort. In: WYRZYKOWSKI, Roman; DONGARRA, Jack; KARCZEWSKI, Konrad; WASNIEWSKI, Jerzy (eds.). *Parallel Processing and Applied Mathematics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 403–410. ISBN 978-3-642-14390-8.
14. AL-HAJ BADDAR, Sherenaz W.; BATCHER, Kenneth E. *Designing Sorting Networks: A New Paradigm*. 1st 2011;1. Aufl.; New York, NY: Springer New York, 2011. ISBN 9781461418511.
15. HOARE, Charles A. R. Algorithm 64: Quicksort. *Commun. ACM*. 1961, vol. 4, no. 7, p. 321. ISSN 0001-0782. Available from DOI: `10.1145/366622.366644`.
16. HOARE, Charles A. R. Quicksort. *The Computer Journal*. 1962, vol. 5, no. 1, pp. 10–16. ISSN 0010-4620. Available from DOI: `10.1093/comjnl/5.1.10`.
17. STEIN, Clifford; LEISERSON, Charles E.; CORMEN, Thomas H.; RIVEST, Ronald L. *Introduction to algorithms*. Quicksort. 2009. ISBN 0262033844.
18. SINGLETON, Richard C. Algorithm 347: An Efficient Algorithm for Sorting with Minimal Storage [M1]. *Commun. ACM*. 1969, vol. 12, no. 3, pp. 185–186. ISSN 0001-0782. Available from DOI: `10.1145/362875.362901`.
19. BLUM, Avrim. *Probabilistic Analysis and Randomized Quicksort* [online]. 2011 [visited on 2021-05-05]. Available from: `https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0906.pdf`.
20. BLELLOCH, Guy E. *Synthesis of Parallel Algorithms*. Prefix sums and their applications. Morgan Kaufmann, 1993.

21. SEDGEWICK, Robert. Implementing Quicksort Programs. *Commun. ACM*. 1978, vol. 21, no. 10, pp. 847–857. ISSN 0001-0782. Available from DOI: 10.1145/359619.359631.

22. GOOGLE LLC. *GoogleTest* [comp. software]. 2019. Version v1.10.0 [visited on 2021-03-20]. Available from: https://github.com/google/googletest.

23. EVANS, D.J.; DUNBAR, R.C. The parallel quicksort algorithm part i–run time analysis. *International Journal of Computer Mathematics*. 1982, vol. 12, no. 1, pp. 19–55. Available from DOI: 10.1080/00207168208803323.

24. BOŽIDAR, Darko. *Vzporedni algoritmi za urejanje podatkov*. Ljubljana, 2015. MA thesis. University of Ljubljana, Faculty of Computer and Information Science. Supervised by doc. dr. Tomaž DOBRAVEC.

25. *GNU C++ Standard Library Documentation* [online]. 2009 [visited on 2021-05-05]. Available from: https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01027.html. [software].

26. NVIDIA. *NVIDIA Nsight Systems* [online]. 2021. Version 2021.2 [visited on 2021-05-12]. Available from: https://developer.nvidia.com/nsight-systems.

27. *MPI* [online]. [N.d.] [visited on 2021-05-12]. Available from: https://www.mpi-forum.org/docs/. [software].

Appendix A

Acronyms

CDP CUDA Dynamic Parallelism

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

GPU Graphics Processing Unit

SIMT Single Instruction, Multiple Threads

STL Standard Template Library

TNL Template Numeric Library

Appendix B

Contents of enclosed medium

- `readme.txt` ..... the file with medium contents description
- `src` ..... the directory of source codes
  - `GPUSort` ..... implementation sources
  - `measuring` ..... the directory of measurement source codes
  - `otherGPUSorts` ..... the directory of other implementations
- `text` ..... the directory of LaTeX source codes of the thesis
- `extra` ..... extra pictures and tables of measurements
- `thesis.pdf` ..... the thesis text in PDF format