WWU MÜNSTER

University of Münster
School of Business & Economics

Tobias Zimmermann

Implementation of parallel map, zip, and reduce operations using CUDA for high performance computing

Seminar Thesis

in the context of the seminar "Parallel Programming"

at the Chair for Practical Computer Science
(University of Münster)

Principal Supervisor:	Prof. Dr. Herbert Kuchen
Associate Supervisor:	Nina Herrmann, M.Sc.
Presented by:	Tobias Zimmermann [454407] Rudolf-Harbig-Weg 2a 48149 Münster +49 170 1250298 t_zimm11@uni-muenster.de
Submission:	30th May 2023

Contents

Figures

Tables

Listings

1 Introduction

Nowadays, computer programs process huge amounts of data. The operations performed on these data are usually not that complex or computation heavy. Since GPUs were designed to render graphics which needs a high number of simple parallel computations, they became really good at that kind of task. Thus, in the early 2000s, the general purpose computing on graphics processing units (GPGPU) came up. Computing on GPUs works seamlessly good, especially for data parallel operations that operate simple functions on data that can be applied independently. Among these operations, the map, zip, and reduce operations are commonly used, especially in the context of data science cf. [Ans22; Akh+13].

Due to this, this seminar thesis discusses the implementation of parallel map, zip and reduce operations with GPUs using CUDA. For this, the theoretical foundations needed during this thesis will be presented in chapter two. Here, related works in the field of parallel algorithmic skeletons and data parallel operations will be presented and the foundations for programming on GPUs with CUDA will be explained. In chapter three the sequential and parallel implementation of the map, zip and reduce operations are presented with a clear focus on the parallel implementation and techniques used. Followed by a runtime comparison in chapter four, where both implementations run in an experimental setup on a high performance computing cluster. Finally, a Conclusion will wrap up this seminar thesis.

2 Foundations

This chapter consists of theoretical foundations needed during this thesis. First, related work is presented on algorithmic skeletons for parallel computing in general and data parallel operations like map, zip, and reduce in detail. After that, foundations of CUDA that is used for the parallel implementation in this thesis will be covered. By taking a look into the key features and characteristics of CUDA and programming on NVIDIA GPUs, followed by taking a look at the architecture and memory management in CUDA.

2.1 Related work

In this section, related work about parallel algorithmic skeletons and data parallel operations will be covered.

2.1.1 Algorithmic skeletons

Parallel programming requires low level knowledge about the programming language and hardware used during the development. Thus, algorithmic skeletons can be used as high level programming constructs to abstract the recurring low level patterns of parallel computation. By using algorithmic skeletons, parallel programming is simplified for the developers and common mistakes can be avoided, enabling developers to reuse common and optimized implementations of parallel problems cf. [EK10; GC11].

Especially in multicore and multi GPU systems, the usage of algorithmic skeletons becomes very beneficial. This is because the complexity of orchestrating and managing multiple CPU chores and GPUs in a system becomes very tedious, especially paired with a high amount of data to be distributed among this system. Thus, by using algorithmic skeletons, the development of parallel programs in such systems becomes way more efficient and less error-prone. By taking care of recurring parallel computation patterns and communication among the participating hardware, high level libraries like SkePU, SkelCL or Muesli provide algorithmic skeletons for common parallel computations and used data types. Thereby hiding the low level parallel implementations, data management and communication from the developer cf. [Ern+21; SG14; CPK09].

Furthermore, algorithmic skeletons work generally well together with the concept of higher order functions, coming from the field of functional programming. Com-

bined with the concepts of generics and templates from the field of object-oriented programming, this allows for a high level of abstraction cf. [GC11; KS02].

Last but not least, algorithmic skeletons in context of distributed computing can be used to ensure reliability and efficiency of implementations while also being portable and platform independent. Thus, they can be used to design distributed parallel systems, providing a high level structured approach of developing such systems cf. [EK12a; EK12b].

2.1.2 Data parallel operations

At data parallelism, data gets distributed among different nodes for parallel computation operating simultaneously. Data parallel operations are a highly effective parallelization for common types of data manipulation, particular when used on large datasets. These operations conclude of specific operations such as the map, zip and reduce operation which are the focus of this thesis. The map operation applies a function to a given data set element wise, while the zip operation combines two datasets into a single dataset also applying a function to each pair of elements. The reduce operation on the other hand reduces a dataset to a single value applying an aggregation function such as the summation of the data set. The operations, among others, are the foundation of many data parallel algorithms and can be efficiently implemented in parallel cf. [Akh+13; EK10].

2.2 CUDA Foundations

In this section, foundations of CUDA and programming on NVIDIA GPUs will be shown. First, the key features and characteristics of CUDA will be examined and after that the architecture and memory management while programming on a GPU with CUDA will get explained.

2.2.1 Key features and characteristics

CUDA or the Compute Unified Device Architecture is an application programming interface model developed by NVIDIA. It enables developers to use NVIDIA GPUs for general purpose processing/computing and, unlike alternatives like OpenCL, is optimized for NVIDIA GPUs. While on a CPU threads are typically heavyweight and in a low quantity, on a GPU threads are lightweight and can be executed in thousands in parallel. This makes GPUs ideal for data parallel tasks, where a lightweight operation is performed many times on a big amount of data cf. [Che14].

Parallel programming with a GPU is heterogeneous computing, as a GPU does not run on its own but needs a CPU. There are two parts of instructions when programming with CUDA. Namely, the implementation can be divided into host instructions and device instructions. The host is the CPU, while the GPU is called the device. The heart of any CUDA program is the kernel function. A kernel function is the function that gets called on the host to start an execution of this kernel function in the device. The kernel functions are written in standard C or C++ with some extensions and restrictions. When calling the kernel function from the host, it is called with a grid and block size. The threads in CUDA are organized in blocks, while the blocks themselves are organized in grids. A kernel always launches one grid, consisting of as many as blocks defined in the kernel call, which consists of as many threads as launched per block. The threads of different blocks cannot communicate and rely on unique coordinates, namely the block index (in a grid) and thread index (in a block). These coordinates can be up to three-dimensional cf. [Ans22; Che14].

2.2.2 Architecture and Memory management

NVIDIA GPUs most basic unit is the simple compute core, which is capable of small operations. Groups consisting of 32 simple compute cores are called processing blocks. The executing threads of a CUDA kernel are grouped on groups consisting of 32 threads each, which are also called warps. Thus, in [Ans22] the processing blocks are also called warp-engines, which is the basic execution unit in a CUDA kernel program. These are then again grouped to symmetric multiprocessors (SM) that also contain a register, shared memory and L1 cache shared between the warp-engines. One GPU then consists of multiple grouped symmetric multiprocessors and also provide a L2 cache shared by all of them cf. [Ans22].

The GPU consists of different types of memory which are organized hierarchically. Not all of them are relevant for this thesis, so they will not be described. The main memory is where the host can write data to and read it from, which is relatively slow compared to the other memory types. It can be reused by following kernel function calls, also asynchronous transfers are possible. The constant memory has a dedicated cache, which makes it very fast for values accessed by all threads in a warp but is limited due to its size of 64 KB. The shared memory as described above is shared within an SM. The size required can be declared at compile or kernel launch time. Due to the latency, the shared memory is best used for communication in a thread block. A CUDA program usually consists of first copying data from the host to the device memory, then performing operations on the data using kernels and

finally copying the data back to the host memory from the device. The memory should first be allocated on the device before copying it. After the memory is not needed any more on the GPU, it should be freed up cf. [Che14; Ans22; NVI22].

This concludes the chapter about the theoretical foundations needed for this thesis. In the following the implementations of map, zip and reduce operations will be presented. For this, the various concepts mentioned above will be used. Followed by a Runtime comparison between the sequential and parallel implementations and an analysis of the resulting runtimes.

3 Implementation of the algorithmic skeletons

In this chapter, the implementation of algorithmic skeletons for map, zip and Reduce operations will be presented. First, the sequential implementation in C++ will be shortly introduced. After that, it will be explained how these operations can be implemented in parallel using CUDA and what needs to be considered while these parallel implementations.

3.1 Sequential implementation

This section focuses on the sequential implementation of the algorithmic skeletons for map, zip and Reduce operations.

3.1.1 Sequential map implementation

Listing 1 shows the sequential implementation of the map operation in C++ as an algorithmic skeleton. The implementation makes use of templates for the data types. In that way, the map operation can be applied on a vector of any data type. The function iterates through the input vector elementwise. In each step, it applies the given function to the input data and stores the result in the output vector cf. [Int20].

```
1 template<typename T, typename F>
2 void map(std::vector <T>& input, std::vector <T>& output, F func){
3     int n  = input.size();
4     for (int i = 0; i < n; i++){
5         output[i] = func(input[i]);
6     }
7 }
```

Listing 1 Sequential map implemenation in C++

3.1.2 Sequential zip implementation

Similar to the sequential implementation of the map operation, the zip operation can be implemented. As shown in listing 2 the algorithmic skeleton for the zip operation, it also iterates over each element in the input vector. But in contrast to the sequential implementation of the map operation, the skeleton takes two input vectors. To avoid accessing an index bigger than the vector size, it iterates over the length of the smaller input vector. For each step, it applies the given function

pairwise to the current element of both input vectors. The result of the function is then stored in the output vector cf. [Int20].

```
1 template<typename T1, typename T2, typename T3, typename F>
2 void zip(std::vector <T1>& input1, std::vector <T2>& input2, std::
      vector <T3>& output, F func){
3      int n = std::min(input1.size(), input2.size());
4      for (int i = 0; i < n; i++){
5          output[i] = func(input1[i], input2[i]);
6      }
7 }
```

Listing 2 Sequential zip implemenation in C++

3.1.3 Sequential Reduce implementation

The sequential implementation for an algorithmic skeleton for the reduce operation takes one input vector just like the map operation. But the first element of the input vector is directly stored in the output variable. Then it iterates over the rest of the input vector elementwise. For each following element of the input vector, it applies the given function to the current output variable and the current element. The result of this function is then again stored in the output variable. Step by step, reducing the input vector to one single value.

```
1 template<typename T, typename F>
2 void reduce(std::vector <T>& input, T& output, F func){
3      int n = input.size();
4      output = input[0];
5      for (int i = 1; i < n; i++){
6          output = func(output, input[i]);
7      }
8 }
```

Listing 3 Sequential Reduce implemenation in C++

3.2 Parallel implementation

In this section, the parallel implementation of algorithm skeletons for map, zip and reduce operations will be presented. The parallel implementation is done using C++ and CUDA, and the usage of CUDA for this purpose is explained in this section.

3.2.1 Parallel map and zip implementation

In contrast to the sequential implementation, the parallel implementation of the algorithmic skeleton for the map operation is split into two parts. The map function that is called by the user and the kernel function.

First, the function called by the user, that is shown in listing 4 will be explained. Just like in the sequential implementation, the user calls the function with a vector containing the input data of any give data type. Furthermore, the user provides a vector where the output should be stored together with the function that should be applied in the map operation. In contrast to the sequential implementation, the user also provides a number of threads, that determines the number of executions happening in parallel. For the parallel computation on a GPU, the data has to be transferred to the device before any operation on the data can take place. To do so, the GPU memory gets allocated in lines six and seven for the input vector and the output vector, respectively. After that, the data of the input vector gets copied from the host to the corresponding allocated memory on the device in line eight. In line nine, the block dimension is initialized with the number of threads, so that the size of a block of threads on the GPU is equal to the number of threads to be used. The number of blocks needed to cover the size of the data is calculated in line 10. By adding one less than the number of threads per block before dividing by this number of threads per block, it is ensured that enough blocks are used in case the size of the data is not a multiple of the block size. After that, the kernel function is called with the dimensions and pointers to the GPU memory from the previous steps. The kernel function gets executed by the given number of threads multiplied by the number of grids. After the application of the kernel function on the GPU has finished, the map operation is done but the results of it is still stored on the GPU or device memory. Thus, in order for the map function to finish, these results need to be transferred from the device back to the host in line twelve. Finally, after freeing up the previously allocated memory, the function has finished cf. [Ans22; Che14; NVI22].

After looking at the function called by the user, we can now take have a look at the kernel function that is the heart of the algorithmic skeleton. The kernel function gets called during the map function and can be seen in listing 5. The *"__global__"* keyword is needed for the CUDA compiler to know that this is a function to be executed on the device but called from the host. Whereas, the given function to be applied on the input data needs to be decorated with the *"__device__"* keyword, which let the CUDA compiler know that this functions is to be executed on the

```
1 template<typename T, typename F>
2 void map(std::vector <T>& input, std::vector <T>& output, F func,
      int numThreads){
3     int size = input.size();
4     T* d_input;
5     T* d_output;
6     cudaMalloc(&d_input, size * sizeof(T));
7     cudaMalloc(&d_output, size * sizeof(T));
8     cudaMemcpy(d_input, input.data(), size * sizeof(T),
          cudaMemcpyHostToDevice);
9     dim3 dimBlock(numThreads);
10    dim3 dimGrid((size + dimBlock.x - 1) / dimBlock.x);
11    map_kernel<<<dimGrid, dimBlock>>>(d_input, d_output, size, func
          );
12    cudaMemcpy(output.data(), d_output, size * sizeof(T),
          cudaMemcpyDeviceToHost);
13    cudaFree(d_input);
14    cudaFree(d_output);
15 }
```

Listing 4 Parallel Map Skeleton implemenation in CUDA

device and also called from it. At the start of each thread, the unique index in the entire grid of blocks of the GPU is calculated in line three. This is important because each and every thread should operate on a different element of the input vector. In case that there are more threads started than there are elements in the input vector, these threads should not do anything, so this is checked in line four. However, every other thread equally to the operation inside the loop of the sequential implementation should apply the given function to its corresponding element of the input data and store the result at the corresponding index of the output data cf. [Ans22; NVI22].

```
1 template <typename T, typename O, typename F>
2 __global__ void map_kernel(T* input, O* output, int size, F func){
3     int idx = blockIdx.x * blockDim.x + threadIdx.x;
4     if (idx < size){
5         output[idx] = func(input[idx]);
6     }
7 }
```

Listing 5 Parallel map kernel implemenation in CUDA

As the parallel implementation of the algorithmic skeleton for the zip operation is pretty similar to the map operation, it will not be presented in detail, but can be found in appendix A. Unlike in the map operation, here as in the sequential implementation, two input vectors are given by the users and the given function is applied pairwise on the input vectors. The rest works in the same way as explained above for the map operation.

3.2.2 Parallel Reduce implementation

The parallel implementation of the reduce operation requires more sophisticated techniques. Because the reduce function that is called by the user is pretty similar to the map and zip operation, it is not explained in detail and not shown here, but can also be found in appendix A. Though, some differences should be noted. Unlike in the other implementations, the reduce implementation consists of two pointers for the output. One pointer for intermediate results and one pointer for the final output. This is because the reduce operation is first performed for each block on the GPU and after this is done, the reduce operation is performed again on the intermediate results to reduce them further to the final result cf. [Che14; Har07].

In listing 6 the kernel function for the parallel reduction is shown. The shared memory declared with the *"__shared__"* keyword is used to store intermediate results of the reduce operation. It is only accessible by the threads within a block. For that, first the input data is loaded into the shared memory within lines six to eleven. If there is no corresponding input value, because there are more threads than elements in the input vector, the zero value is stored in the shared memory. The *"__syncthreads();"* in line twelve ensures, that all threads have finished writing the input data into the shared memory before proceeding cf. [Har07; Che14].

```
1  template <typename T, typename O, typename F>
2  __global__ void reduce_kernel(T* input, O* output, int size, F func
       ){
3      __shared__ O sdata[1024];
4      unsigned int tid = threadIdx.x;
5      unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
6      if (idx < size){
7          sdata[tid] = input[idx];
8      }
9      else {
10         sdata[tid] = 0;
11     }
12     __syncthreads();
13     for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1){
14         if (tid < s){
15             sdata[tid] = func(sdata[tid], sdata[tid + s]);
16         }
17         __syncthreads();
18     }
19     if (tid == 0){
20         output[blockIdx.x] = sdata[0];
21     }
22 }
```

Listing 6 Parallel reduce kernel implemenation in CUDA

In the loop from line 13 to 18, a divide and conquer strategy is used to pairwise apply the given function to pairs of the shared memory. After that, in line 17 it is again ensured that all threads have finished their step of the reduce operation before starting with the next one. At the end of the reduce kernel in lines 19 and following, only the first thread of each block stores the result of this block to the corresponding output index cf. [Har07].

It should be noted that this implementation of the parallel reduce operation can be further optimized. For example, by applying a first reduction during the load of the input data, unrolling the last warp and more sophisticated techniques as it can be seen in [Har07] and explained in more detail in [Che14]. Also, the size of the shared memory could be defined externally at the kernel call instead of being directly given as in listing 6.

This concludes the chapter about the sequential and parallel implementation of the map, zip and reduce operation. In short, we have seen how these operations can be implemented in a sequential manner using C++ and how they can be transformed to parallel implementations using C++ and CUDA. In the next chapter, these implementations will be benchmarked on a high performance cluster, and what has been gained by parallelising the different operations will be analysed.

4 Runtime comparison and analysis

This chapter is about the run time comparison between the sequential and parallel implementations from the last chapter. Furthermore, the contribution of different parts of the parallel implementations will be analysed to discover further optimisation potential. Starting with the experimental setup, the results of the runtime comparison will be presented, followed by an analysis of the results.

4.1 Experimental setup

For the runtime comparison between the sequential and parallel implementations of map, zip and reduce operations, the HPC cluster Palma II [WWUa] at the university of Münster will be used. For performance reasons, the implementations need to be compiled on the machines they will run on. For this reason, CMake [Kit] will be used to compile the implementations during the runtime comparison. For this, a *"CMakeLists"* needs to be delivered, defining the corresponding C++ and CUDA versions to use, what flags to use for compilation, what to include and what to use as main executable. After testing the build of the implementations locally before running the runtime comparison on the HPC cluster, it has to be tested and eventually debugged on the corresponding partition of the cluster, where the comparison should run on.

If the implementations build and run successfully on the partition of the HPC cluster, a job to be automatically run on the partition can be defined. For this, a shell script is used to define the job file. The shell scripts, defining the jobs for the runtime measurement of the sequential and parallel implementation respectively, can be found in appendix B. In both jobs, first using the *"SBATCH"* keyword are used to configure Slurm [Slu] the workload manager for jobs to be run on the HPC cluster. For example, it is defined that the job should run exclusively on the partition that is important to get comparable results. Also, what partition to use and how many nodes, CPUs and GPUs is defined here. For comparability reasons, both jobs run on the *"gpu2080"* partition and use one node and one CPU, while the parallel implementation also uses one GPU. After the Slurm configuration and loading the required modules for the corresponding job, the implementations are compiled with CMake as explained above cf. [WWUb].

The *"gpu2080"* partition used in this setup uses a Zen3 (EPYC 7513) CPU and a GeForce RTX 2080 Ti GPU. It consists of five nodes and eight GPUs per node, but only one node and one GPU is used during this runtime comparison cf. [WWUb].

Now, after the implementations are compiled successfully, each runtime experiment is run for 50 times. At each run, the program gets executed ten times with different vector sizes from 10 to 100 million elements, increased by 10 million at each step. The output of the program is stored in one file for each run, in a directory for the implementation. The output of the program contains the time measurements in a CSV format, also at the start of each run a header in CSV format is printed on top of the output file. For time measurement in the sequential implementation, the time for each operation is measured in milliseconds. In contrast to this, in the parallel implementation, time measurement takes not only place for every operation, but also for the different parts of the operation. Namely, transferring the data from host to the device, duration of the kernel function, transferring the output data from device back to host and the overall duration. This happens to keep track of the contributions of the different parts to the overall duration of the operation.

4.2 Results

To achieve the final results of the runtime comparison, for both implementations, the first five runs were ignored for the device to warm up. After that, the mean over the other 45 runs was calculated. The overall time for the map, reduce and zip operation and each implementation can be seen in figure 1. The runtime is given in seconds and the size of the input vectors is given in ten million. The blue line shows the sequential runtime, while the parallel runtime is shown by the orange line in each graph.

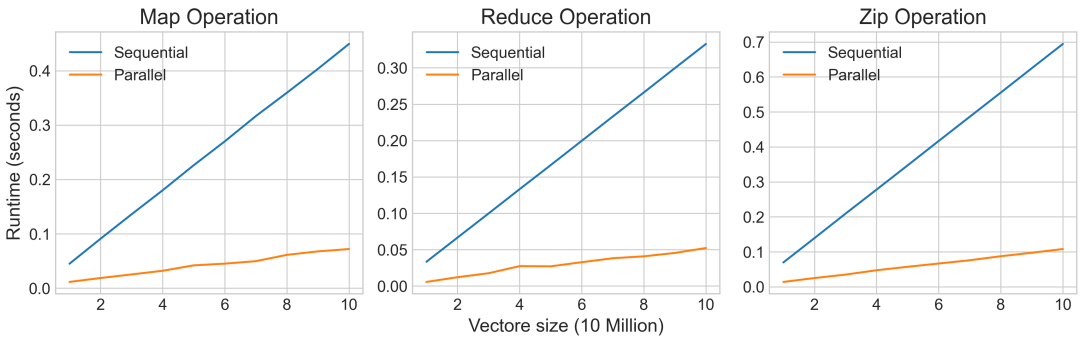

Figure 1 Runtime of sequential and parallel implementaions for different data sizes

For the parallel implementation, the contribution of the different parts of the operations can be seen in figure 2. It shows the runtime in milliseconds and the vector size in ten million. The blue bar shows the time it needs to copy the input data

from the host to the device, while the green bar shows how long it takes to transfer the data back from device to host, respectively. The orange bar corresponds to the time the kernel needs to perform the core operation. It should be noted that the time that is needed to transfer the output of the reduce operation back to the host is too small to be seen, namely around 0.01 milliseconds.

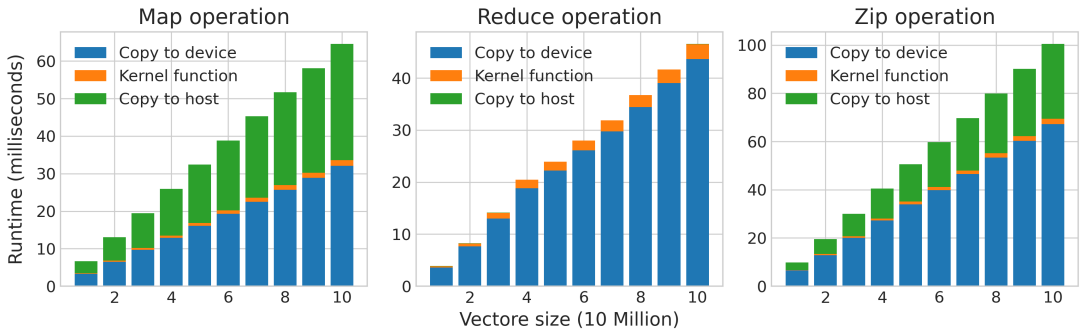

Figure 2 Contribution of different parts of the operations to the parallel runtime for different data sizes

4.3 Analysis

As seen above, the parallel implementation is fairly faster than the sequential implementation for every operation, as expected. In figure 1 it can be noticed that even for the biggest vector size of 100 million, the runtime of the parallel implementation is barely above the runtime for the smallest vector size of the sequential implementation. For the map and zip operation, this is due to their nature of being embarrassingly parallel. This means that there is no dependency between the steps of the operations, and so there is no need for communication. The reduce operation on the other hand depends on the results of further steps. In this case, the parallel speedup is achieved by using a divide and conquer strategy as described in the last chapter cf. [BT15].

	Sequential	**Parallel**	**Speedup**
Map	2,48s	0,42s	5,83x
Reduce	1,83s	0,30s	6,13x
Zip	3,82s	0,61s	6,24x
Total	8,13s	1,34s	18,20x

Table 1 Runtimes and speedup of map, reduce and zip operation

In table 1 the overall runtimes and speedup of the map, zip and reduce operation can be seen. Each parallel implementation is around six times faster than the corresponding sequential implementation. The map operation achieves the smallest speedup while the zip operation can achieve the highest speedup in this benchmark. Overall, the parallel implementations are together around 18 times faster than the sequential implementations.

	Sequential	**Parallel**	**Speeup**
Map	2,48	0,01	292,82
Reduce	1,83	0,02	107,42
Zip	3,82	0,01	316,45
Total	8,13	0,04	716,69

Table 2 Runtimes and speedup of map, reduce and zip kernel operation

It can be seen in figure 2 the time needed to execute the kernels is relatively small compared to the time needed for transferring the data from host to device and vice versa. Thus, the transfer of the data becomes a bottleneck in this case. The speedup for the runtimes needed by the kernels compared to the sequential implementation is shown in table 2. Here, an overall speedup of around 700 times can be achieved, with around 300 for the map and zip operation and around 100 for the reduce operation. This also underlines the nature of the operations, as mentioned above.

As the data transfer becomes a bottleneck, this could be further optimized by spreading the data across multiple GPUs in parallel. By utilizing parallel processing on the CPU to spread the data among the GPUs overall, one could achieve a much faster execution time. Of course, this further optimization would come at the cost of more resource consumption.

5 Conclusion

After presenting the foundations needed for this thesis in chapter two, the implementation of the map, zip and reduce operation sequentially as well as in parallel was explained. As limitation, it should be noticed that the reduce operation was not fully optimized as shown in [Har07] due to complexity and scope of this thesis. This also reflects in table 2 as the speedup for the reduce operation is only one third of the other operations. Furthermore, the runtime was only evaluated for different amount of data sizes to analyse the speedup of the implementations. In addition to that, also the runtimes for different amounts of threads should be analysed. Last but not least, as can be seen in figure 2 the data transfer between host and device becomes a bottleneck compared to the kernel timings of the operations. Thus, further optimization of the operations by scaling among multiple GPUs or even multiple computing nodes should be examined.

All in all, with the parallel implementations, a decent speedup could be achieved as seen tabled 1 and 2. Allowing for more data processing in a lot shorter time. These example implementations [1] were good for educational reasons. But as explained in chapter two, especially for such basic operations the usage of algorithmic skeleton libraries like [Ern+21; SG14; CPK09] should be considered since they are less error-prone, optimized for multiple hardware setups and also allow for multiple GPUs and nodes to be used, highly reducing the development overhead.

[1] The whole implementation is submitted with this seminar thesis

Appendix

A Parallel implementations

A.1 Parallel Zip implementation

```
1 template <typename T1, typename T2, typename O, typename F>
2 __global__ void zip_kernel(T1* input1, T2* input2, O* output, int
      size, F func){
3     int idx = blockIdx.x * blockDim.x + threadIdx.x;
4     if (idx < size){
5         output[idx] = func(input1[idx], input2[idx]);
6     }
```

Listing 7 Parallel zip kernel implemenation in CUDA

```
1 template<typename T1, typename T2, typename T3, typename F>
2 void zip(std::vector <T1>& input1, std::vector <T2>& input2, std::
      vector <T3>& output, F func, int numThreads){
3     int size = std::min(input1.size(), input2.size());
4     T1* d_input1;
5     T2* d_input2;
6     T3* d_output;
7     cudaMalloc(&d_input1, size * sizeof(T1));
8     cudaMalloc(&d_input2, size * sizeof(T2));
9     cudaMalloc(&d_output, size * sizeof(T3));
10    cudaMemcpy(d_input1, input1.data(), size * sizeof(T1),
          cudaMemcpyHostToDevice);
11    cudaMemcpy(d_input2, input2.data(), size * sizeof(T2),
          cudaMemcpyHostToDevice);
12    dim3 dimBlock(numThreads);
13    dim3 dimGrid((size + dimBlock.x - 1) / dimBlock.x);
14    zip_kernel<<<dimGrid, dimBlock>>>(d_input1, d_input2, d_output,
          size, func);
15    cudaMemcpy(output.data(), d_output, size * sizeof(T3),
          cudaMemcpyDeviceToHost);
16    cudaFree(d_input1);
17    cudaFree(d_input2);
18    cudaFree(d_output);
19 }
```

Listing 8 Parallel Zip Sekeleton implemenation in CUDA

A.2 Parallel Reduce implementation

```
1 template<typename T, typename F>
2 void reduce(std::vector <T>& input, T& output, F func, int
      numThreads){
3     int size = input.size();
4     T* d_input;
5     T* d_output;
6     T* d_final_output;
7     cudaMalloc(&d_input, size * sizeof(T));
8     cudaMalloc(&d_output, sizeof(T) * ((size + 1023) / 1024));
9     cudaMalloc(&d_final_output, sizeof(T));
10    cudaMemcpy(d_input, input.data(), size * sizeof(T),
          cudaMemcpyHostToDevice);
11    dim3 dimBlock(numThreads);
12    dim3 dimGrid((size + dimBlock.x - 1) / dimBlock.x);
13    reduce_kernel<<<dimGrid, dimBlock>>>(d_input, d_output, size,
          func);
14    reduce_kernel<<<1, dimGrid.x>>>(d_output, d_final_output,
          dimGrid.x, func);
15    cudaMemcpy(&output, d_final_output, sizeof(T),
          cudaMemcpyDeviceToHost);
16    cudaFree(d_input);
17    cudaFree(d_output);
18    cudaFree(d_final_output);
19 }
```

Listing 9 Parallel Reduce Skeleton implemenation in CUDA

B Runtime Experiment

B.1 Sequential jobfile

```
#!/bin/bash
#SBATCH --exclusive
#SBATCH --partition=gpu2080
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --gres=gpu:0
#SBATCH --time=2:00:00
#SBATCH --job-name=sequential_runtime_comparison
#SBATCH --output=/scratch/tmp/t_zimm11/gpu2080node1_sequential.out
#SBATCH --error=/scratch/tmp/t_zimm11/gpu2080node1_sequential.error
#SBATCH --mail-type=ALL
#SBATCH --mail-user=t_zimm11@uni-muenster.de
#SBATCH --mem=0
partition=gpu2080
implementation="sequential"
dirname=$(date +"%Y-%m-%dT%H-%M-%S-${partition}-${implementation}")
module purge
ml foss/2022a
ml CMake/3.23.1
cd /home/t/t_zimm11/
path=/scratch/tmp/t_zimm11/${dirname}
mkdir -p "$path"
buildname=build-${partition}
(
cd $implementation
rm -rf "$buildname"
mkdir "$buildname"
cd "$buildname"
cmake ..
cmake --build .
)
for (( run=1; run<=50; run++ )) do
    echo "size,map_duration,reduce_duration,zip_duration" > "${path
       }/ppseminar_sequential_${run}.out"
    for (( size=10000000; size<=100000000; size+=10000000 )) do
        ./${implementation}/${buildname}/main $size >> "${path}/
           ppseminar_sequential_${run}.out"
    done
done
exit 0
```

Listing 10 Sequential jobfile

B.2 Parallel jobfile

```
#!/bin/bash
#SBATCH --exclusive
#SBATCH --partition=gpu2080
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --gres=gpu:1
#SBATCH --time=1:00:00
#SBATCH --job-name=parallel_runtime_comparison
#SBATCH --output=/scratch/tmp/t_zimm11/gpu2080node1_parallel.out
#SBATCH --error=/scratch/tmp/t_zimm11/gpu2080node1_parallel.error
#SBATCH --mail-type=ALL
#SBATCH --mail-user=t_zimm11@uni-muenster.de
#SBATCH --mem=0
partition=gpu2080
implementation="parallel"
dirname=$(date +"%Y-%m-%dT%H-%M-%S-${partition}-${implementation}")
module purge
ml palma/2022a
ml CUDA/11.7.0
ml foss/2022a
ml UCX-CUDA/1.12.1-CUDA-11.7.0
ml CMake/3.23.1
cd /home/t/t_zimm11/
path=/scratch/tmp/t_zimm11/${dirname}
mkdir -p "$path"
buildname=build-${partition}
(
# Same as above
)
for (( run=1; run<=50; run++ )) do
    echo "size,numThreads,map_copy_device,map_kernel,map_copy_host,
        map_total,map_total_chrono,reduce_copy_device,reduce_kernel,
        reduce_copy_host,reduce_total,reduce_total_chrono,
        zip_copy_device,zip_kernel,zip_copy_host,zip_total,
        zip_total_chrono" > "${path}/ppseminar_parallel_${run}.out"
    for (( size=10000000; size<=100000000; size+=10000000 )) do
        ./${implementation}/${buildname}/main $size >> "${path}/
            ppseminar_parallel_${run}.out"
    done
done
exit 0
```

Listing 11 Parallel jobfile

References

[Akh+13] Darkhan Akhmed-Zaki et al. “Design of Distributed Parallel Computing Using by MapReduce/MPI Technology”. In: *Parallel Computing Technologies*. Ed. by Victor Malyshkin. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 139–148. DOI: 10.1007/978-3-642-39958-9_12.

[Ans22] Richard Ansorge. *Programming in Parallel with CUDA: A Practical Guide*. 1st ed. Cambridge University Press, May 31, 2022. DOI: 10.1017/9781108855273. URL: https://www.cambridge.org/core/product/identifier/9781108855273/type/book (visited on 05/12/2023).

[BT15] James Brodman and Peng Tu, eds. *Languages and Compilers for Parallel Computing: 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers*. Vol. 8967. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015. DOI: 10.1007/978-3-319-17473-0. URL: https://link.springer.com/10.1007/978-3-319-17473-0 (visited on 05/28/2023).

[Che14] John Cheng. *Professional Cuda C programming*. Wrox programmer to programmer. Indianapolis, IN: John Wiley and Sons, Inc, 2014. 497 pp.

[CPK09] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. “The Münster Skeleton Library Muesli: A comprehensive overview”. In: (2009).

[EK10] Johan Enmyren and Christoph W. Kessler. “SkePU: a multi-backend skeleton programming library for multi-GPU systems”. In: *Proceedings of the fourth international workshop on High-level parallel programming and applications*. HLPP ’10. New York, NY, USA: Association for Computing Machinery, Sept. 25, 2010, pp. 5–14. DOI: 10.1145/1863482.1863487. URL: https://dl.acm.org/doi/10.1145/1863482.1863487 (visited on 05/29/2023).

[EK12a] Steffen Ernsting and Herbert Kuchen. “Algorithmic skeletons for multi-core, multi-GPU systems and clusters”. In: *International Journal of High Performance Computing and Networking* 7.2 (2012), p. 129. DOI: 10.1504/IJHPCN.2012.046370. URL: http://www.inderscience.com/link.php?id=46370 (visited on 05/29/2023).

[EK12b] Steffen Ernsting and Herbert Kuchen. “Data Parallel Skeletons for GPU Clusters and Multi-GPU Systems”. In: *Applications, Tools and Techniques on the Road to Exascale Computing* (2012). Publisher: IOS Press, pp. 509–518. DOI: 10.3233/978-1-61499-041-3-509. URL: https://ebooks.iospress.nl/doi/10.3233/978-1-61499-041-3-509 (visited on 03/09/2023).

[Ern+21] August Ernstsson et al. “SkePU 3: Portable High-Level Programming of Heterogeneous Systems and HPC Clusters”. In: *International Journal of Parallel Programming* 49 (Dec. 1, 2021), pp. 1–21. DOI: 10.1007/s10766-021-00704-3.

[GC11] Sergei Gorlatch and Murray Cole. “Parallel Skeletons”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1417–1422. DOI: 10.1007/978-0-387-09766-4_24. URL: https://doi.org/10.1007/978-0-387-09766-4_24 (visited on 05/29/2023).

[Har07] Mark Harris. “Optimizing parallel reduction in CUDA”. In: *Nvidia developer technology* 2.4 (2007). Publisher: Nvidia Corporation Santa Clara, CA, USA, p. 70.

[Int20] International Organization for Standardization (ISO). *ISO/IEC 14882:2020 Programming languages — C++*. 2020. URL: https://isocpp.org/std/the-standard (visited on 05/26/2023).

[Kit] Kitware, Inc. *CMake*. URL: https://cmake.org/ (visited on 05/28/2023).

[KS02] Herbert Kuchen and Jörg Striegnitz. “Higher-order functions and partial applications for a C++ skeleton library”. In: *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*. JGI02: Joint ACM Java Grande - ISCOPE 2002 Conference (co-located with OOPSLA 2002). Seattle Washington USA: ACM, Nov. 3, 2002, pp. 122–130. DOI: 10.1145/583810.583824. URL: https://dl.acm.org/doi/10.1145/583810.583824 (visited on 05/29/2023).

[NVI22] NVIDIA. *CUDA Toolkit Documentation v12.0 — landing 12.0 documentation*. 2022. URL: https://docs.nvidia.com/cuda/archive/12.0.0/ (visited on 05/29/2023).

[Slu] Slurm Team. *Slurm Workload Manager - Documentation*. URL: https://slurm.schedmd.com/documentation.html (visited on 05/28/2023).

[SG14] Michel Steuwer and Sergei Gorlatch. “SkelCL: a high-level extension of OpenCL for multi-GPU systems”. In: *The Journal of Supercomputing* 69.1 (July 1, 2014), pp. 25–33. DOI: 10.1007/s11227-014-1213-y. URL: https://doi.org/10.1007/s11227-014-1213-y (visited on 03/09/2023).

[WWUa] WWU IT. *High Performance Computing - High Performance Computing - WWU Confluence Wiki*. URL: https://confluence.uni-muenster.de/display/HPC/High+Performance+Computing (visited on 05/28/2023).

[WWUb] WWU IT. *Wiki - High Performance Computing - WWU Confluence Wiki*. URL: https://confluence.uni-muenster.de/display/HPC/Wiki (visited on 05/26/2023).

Declaration of Authorship

I hereby declare that, to the best of my knowledge and belief, this Seminar Thesis titled "Implementation of parallel map, zip, and reduce operations using CUDA for high performance computing" is my own work. I confirm that each significant contribution to and quotation in this thesis that originates from the work or works of others is indicated by proper use of citation and references.

Münster, 30$^{\text{th}}$ May 2023

Tobias Zimmermann

Consent Form

for the use of plagiarism detection software to check my thesis

Last name: Zimmermann **First name**: Tobias
Student number: 454407 **Course of study**: Information Systems
Address: Rudolf-Harbig-Weg 2a, 48149 Münster
Title of the thesis: "Implementation of parallel map, zip, and reduce operations using CUDA for high performance computing"

What is plagiarism? Plagiarism is defined as submitting someone else's work or ideas as your own without a complete indication of the source. It is hereby irrelevant whether the work of others is copied word by word without acknowledgment of the source, text structures (e.g. line of argumentation or outline) are borrowed or texts are translated from a foreign language.

Use of plagiarism detection software The examination office uses plagiarism software to check each submitted bachelor and master thesis for plagiarism. For that purpose the thesis is electronically forwarded to a software service provider where the software checks for potential matches between the submitted work and work from other sources. For future comparisons with other theses, your thesis will be permanently stored in a database. Only the School of Business and Economics of the University of Münster is allowed to access your stored thesis. The student agrees that his or her thesis may be stored and reproduced only for the purpose of plagiarism assessment. The first examiner of the thesis will be advised on the outcome of the plagiarism assessment.

Sanctions Each case of plagiarism constitutes an attempt to deceive in terms of the examination regulations and will lead to the thesis being graded as "failed". This will be communicated to the examination office where your case will be documented. In the event of a serious case of deception the examinee can be generally excluded from any further examination. This can lead to the exmatriculation of the student. Even after completion of the examination procedure and graduation from university, plagiarism can result in a withdrawal of the awarded academic degree.

I confirm that I have read and understood the information in this document. I agree to the outlined procedure for plagiarism assessment and potential sanctioning.

Münster, 30th May 2023

Tobias Zimmermann