Correct Optimized GPU Programs

Mohsen Safari

Correct Optimized GPU Programs

Dissertation

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus
Prof.dr.ir. A. Veldkamp,
on account of the decision of the Doctorate Board,
to be publicly defended
on Wednesday the 6th of July 2022 at 16:45

by

Mohsen Safari

born on the 5th of April, 1991
in Gonbadekavoos, Iran

This dissertation has been approved by:

Prof.dr. M. Huisman (promotor)

DIGITAL SOCIETY INSTITUTE

DSI Ph.D. Thesis Series No. 22-003
Digital Society Institute
P.O. Box 217, 7500 AE
Enschede, the Netherlands

IPA Dissertation Series No. 2022-03
The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Nederlandse Organisatie voor Wetenschappelijk Onderzoek
The work in this thesis was supported by the Mercedes (Maximal Reliability of Concurrent and Distributed Software) project, funded by the NWO grant 639.023.710.

ISBN: 978-90-365-5342-1
ISSN: 2589-7721 (DSI Ph.D. Thesis Series No. 22-003)
DOI: 10.3990/1.9789036553421
Available online at `https://doi.org/10.3990/1.9789036553421`

Typeset with LaTeX
Printed by Ipskamp Printing
Cover design by Ipskamp Printing

© 2022 Mohsen Safari, the Netherlands. All rights reserved. No parts of this thesis may be reproduced, stored in a retrieval system or transmitted in any form or by any means without permission of the author. Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd, in enige vorm of op enige wijze, zonder voorafgaande schriftelijke toestemming van de auteur.

Graduation Committee:

Chairman:	Prof.dr. J.N. Kok
Promotor:	Prof.dr. M. Huisman

Members:

Prof.dr. P. Herber	University of Twente, the Netherlands & University of Münster, Germany
Prof.dr. A.L. Varbanescu	University of Twente, the Netherlands
Dr. B. van Werkhoven	Netherlands eScience Center, the Netherlands
Prof.dr. J.C. Filliâtre	CNRS, France
Prof.dr. A.F. Donaldson	Imperial College London, UK

Acknowledgments

I am writing the acknowledgments as the last part of my PhD thesis. It is an opportunity for me to express my gratitude to many people who helped me during this journey. When I look back at the four years of my PhD, it brings me enjoyable memories.

I express my first gratitude to my supervisor Marieke Huisman. It was a great pleasure to pursue my PhD under your supervision. I really learned many skills from you both on the technical and non-technical part. Many thanks for giving me the freedom to work on the verification of GPU programs. You were always kind, patient and supportive during my PhD. Thank you for allowing me to participate in summer schools and conferences according to my interest. Under your supervision, I experienced an enjoyable PhD period without feeling stress and pressure. I am very grateful that you allowed me to be the organizer of the FMT colloquia. It was a nice experience. Even though we had the tough moments of pandemic, your support and guidance helped me to continue and finish the thesis according to the plan. I could have not finished this thesis without all your assistance.

I express my second gratitude to Jaco van de Pol, who connected me to the Netherlands and the Formal Methods and Tools (FMT) group. I met Jaco at a conference in Tehran, Iran in 2017. At that time I was looking for a PhD position. We had a good discussion after my presentation and he introduced me to Marieke. Thank you Jaco for your support.

I would like to extend my gratitude to the (former and current) secretaries of the group Ida den Hamer and Marion Steenbergen-Boeringa. You were always ready and open to help me during my PhD. Ida, I am sincerely grateful for all your helps, in particular at the beginning of my PhD when I arrived in the Netherlands.

I would like to thank Sebastiaan Joosten and Wytse Oortwijn who kindly assisted me when I joined the group. It was very nice to have discussions and collaborations

with you and I learned many things from both of you!

I am grateful for having nice (ex-)colleagues in the VerCors team: Raúl Monti, Petra van den Bos, Sophie Lathouwers, Lukas Armborst, Pieter Bos, Ömer Şakar, Bob Rubbens, Fauzia Ehsan and Henk Mulder. It was amazing to work with all of you in the project. I would like to thank all my (ex-)colleagues at the FMT group. I had four years of memorable moments with all of you in the group. I really enjoyed the times we had for social activities such as group lunches, social dinners, FMT outgoings and BOCOM!

Rom, it was a pleasure to assist you for Discrete Structures and Efficient Algorithms in module 7. I enjoyed the way you organize it. I am also thankful for taking me to the boxing, even though the pandemic did not allow us to continue it.

Many thanks to all members in my graduation committee: Paula Herber, Ana Lucia Varbanescu, Ben van Werkhoven, Jean-Christophe Filliâtre and Alastair F. Donaldson, for taking time to read my thesis and providing me useful feedback.

I also would like to thank my friends: Masoud Abbasi Alaei, Hiwa Qazi, Sajad Golabi Ghezel Ahmadi, Reza Serajeh, Mohammad Poordaraee and Morteza Mojarradi who came into my life during the PhD time. We had memorable events in our gatherings at the University of Twente.

Last but not least, I would like to express a special gratitude to my family. I would like to thank my parents Jamshid and Masoumeh, my sister Fatemeh and my brother Ali. Thank you for your supports not only during these four years, but my whole life. Being thousands of kilometers away from you, my thoughts are always with you.

Mohsen Safari
Enschede, May 20, 2022

Abstract

Software is widely incorporated in our daily lives. Almost all electronic devices around us, from smartphones and coffee machines to cars and planes are run by computer software.

Thus our lives heavily depends on software, while software is created by humans who make mistakes by nature. Therefore, developing *correct* software becomes a major challenge, where by correctness, we mean how to guarantee that software behaves as we expect. Software correctness is in particular crucial in critical systems (e.g., nuclear power plants, medical devices, etc.) where software failure would threaten human lives and environment; see [LT93, Pra95, LLF+96, DC12, GKTZ12, Wit16].

In addition to reliability and correctness, also software *performance* is demanded by users. Performance is important, in particular, in critical systems where delay in software response would cause serious damage. To increase performance, new hardware and architectures are coming out. According to Moore's law the number of transistors on a CPU chip will double every 18 months. According to Dennard scaling law as transistors become smaller, their power density stays constant. The impact of Moore's and Dennard laws is that manufacturers can raise clock frequency without significantly increasing overall circuit power consumption. Nowadays, Moore's law is not happening anymore by its prediction, and Dennard scaling law is broken down. To address this, multi-core CPUs are invented. However, the number of CPU cores on a machine is limited.

Graphics Processing Units (GPUs) have emerged as co-processors to address this issue and to reduce the performance limitations of data-parallel programs. In contrast to multi-core CPUs, GPUs have hundreds or thousands of cores which are simpler, but more tuned to data parallelism. CUDA and OpenCL are two dominant programming languages for GPUs. Their programming model supports parallel execution, with many threads cooperating together, executing the same

instructions, but on different data (known as Single Instruction Multiple Data (SIMD)).

GPUs are widely-used in industry and they drastically improve the performance of (sequential and multi-threaded) programs. However, errors happen more frequently on GPU programs, because the GPU programming model provides more flexibility to the programmers to control hardware resources. Moreover, also the hierarchical structure of threads and memory layouts make GPU programming more challenging and error-prone. Therefore, proving correctness of GPU programs is of the utmost importance. Concretely, in this thesis we investigate how to prove that a GPU program is data-race free and functionally correct.

Deductive program verification can be used to reason about GPU programs. In a deductive approach, programs are annotated by specifying pre- and postconditions, invariants and intermediate properties written in first-order logic. Then, we use a proof system to verify that the implementation adheres to the specification. However, proving GPU programs using deductive techniques is challenging, because of the many different interleavings of arbitrary threads and the specific programming model of GPU programming. Moreover, program development on GPUs is an iterative process. First, programmers implement a naive version of the algorithm on a GPU. We call this version an unoptimized program, as the focus is on validity of the implementation rather than on the performance. Then, to achieve the most performance out of the GPU, programmers typically apply incremental optimizations manually, tailored to the GPU architecture, resulted in different versions of implementations. These optimizations can be applied manually or semi-automatically at the level of GPU programs prior to compiling the code. Even though there has been some work to automate the process of optimization, there has been little effort to guarantee that correctness is preserved during this optimization process. In addition, automatic correctness preservation of GPU programs is important, to avoid having to manually annotate each new version of the optimized program using deductive approaches.

Therefore, the questions which this thesis addresses is based on two pillars. First, *how to use deductive verification techniques in practice to prove functional correctness of complicated GPU programs*[1]*?* Second, *how to automatically apply GPU optimizations to an unoptimized verified GPU program to guarantee that the final optmized GPU program is still correct?*

This thesis consists of two parts to address the two questions. The first part shows how to leverage deductive verification techniques to reason about complicated GPU programs. Particularly, the first part proves data-race freedom and functional correctness of two implementations of parallel prefix sum algorithms,

[1]By complicated GPU programs, we mean that the GPU implementations of algorithms are not trivial to reason about.

a parallel stream compaction and summed-area table algorithm, and a parallel Bellman-Ford shortest path algorithm. Moreover, this part also shows a generic pattern to prove the permutation property of any (GPU-based) parallel and even (CPU-based) sequential swap-based sorting algorithms. This technique is applied to prove the permutation property of parallel odd-even transposition sort, and sequential bubble sort, selection sort, insertion sort, quick sort, two in-place merge sorts and TimSort.

The second part of the thesis introduces an *annotation-aware (source-to-source) transformations* technique. This means that the transformations should be made *annotation-aware*, i.e., during program transformation, not only the code should be changed automatically, but also the corresponding annotations. The goal is to apply optimization techniques automatically to a verified unoptimized GPU program, step-by-step, with the annotations being transformed automatically, allowing for efficient reverification of the optimized program(s). Concretely, we develop the approach for six optimization techniques, namely loop unrolling, iteration merging, matrix linearization, data prefetching, tiling and kernel fusion to apply them to the verified programs while preserving their provability. As a consequence, this approach results in correct optimized GPU programs. We evaluate this on the same example programs that are verified in the first part of this thesis.

Contents

Acknowledgments **vii**

Abstract **ix**

Contents **xiii**

1 Introduction **1**
1.1 Thesis Outline 5

2 Background **9**
2.1 Deductive Program Verification 9
2.2 GPGPUs 11
2.3 GPGPU Verification Techniques 14
2.3.1 VerCors 17

I Correct GPU Programs; case studies **21**

3 Verification of Parallel Prefix Sums **23**
3.1 Introduction 23
3.2 Prefix Sum Problem 25
3.3 Verification of Blelloch's Parallel Algorithm 25
3.3.1 Blelloch's Prefix Sum 25
3.3.2 Data-Race Freedom 27
3.3.3 Functional Correctness 28
3.4 Verification of Kogge-Stone's Parallel Algorithm 35
3.4.1 Kogge-Stone's Prefix Sum 36
3.4.2 Data-Race Freedom 37

3.4.3 Functional Correctness 38
3.5 Towards CUDA Verification 40
3.5.1 CUDA to PVL Transformation 40
3.5.2 CUDA Verification Challenges 41
3.6 Related Work 43
3.7 Conclusion 44

4 Verification of Parallel Stream Compaction and Summed-Area Table Algorithms 45
4.1 Introduction 45
4.2 Verification of Parallel Stream Compaction Algorithm 46
4.2.1 Stream Compaction Algorithm 47
4.2.2 Data-Race Freedom 48
4.2.3 Functional Correctness 49
4.3 Verification of Parallel Summed-Area Table Algorithm 55
4.3.1 Summed-Area Table Algorithm 55
4.3.2 Data-Race Freedom 56
4.3.3 Functional Correctness 57
4.4 Towards CUDA Verification 61
4.5 Related Work 61
4.6 Conclusion 62

5 Verification of Parallel Bellman–Ford SSSP Algorithm 63
5.1 Introduction 63
5.2 SSSP and the Bellman–Ford Algorithm 64
5.3 Approach 66
5.4 Proof Mechanization 69
5.5 Evaluation and Discussion 74
5.6 Related Work 75
5.7 Conclusion 77

6 Verification of Permutation Property of Sorting Algorithms 79
6.1 Introduction 79
6.2 Permutation Verification of Swap-based Sorting 81
6.2.1 Swap-based Sorting Algorithms 81
6.2.2 Functional Correctness of Swap-based Sorting Algorithms . 81
6.3 Case Studies: Proving Permutation of Swap-based Sorting Algorithms 85
6.3.1 Permutation Verification of Parallel Odd-Even Transposition Sort 87
6.3.2 Permutation Verification of Bubble, Selection and Insertion Sort 89
6.3.3 Permutation Verification of Quick Sort 90

6.3.4 Permutation Verification of Merge Sort . . . 91
6.3.5 Permutation Verification of TimSort . . . 95
6.4 Related Work . . . 95
6.5 Conclusion . . . 96

II Correct GPU Optimizations; annotation transformations 97

7 Annotation-Aware GPU Optimizations 99
7.1 Introduction . . . 99
7.2 Motivational Example . . . 101
7.3 GPU Optimizations . . . 104
7.3.1 Loop Unrolling . . . 104
7.3.2 Iteration Merging . . . 107
7.3.3 Matrix Linearization . . . 109
7.3.4 Data Prefetching . . . 110
7.3.5 Tiling . . . 112
7.3.6 Kernel Fusion . . . 114
7.4 Implementation . . . 123
7.5 Evaluation . . . 125
7.5.1 Experiment Setup . . . 125
7.5.2 Results & Discussion . . . 126
7.6 Related Work . . . 127
7.7 Conclusion . . . 132

8 Conclusions 133
8.1 Contributions . . . 134
8.2 Future Work . . . 135

A Publications by the Author 137

Bibliography 139

Samenvatting 157

CHAPTER 1

Introduction

Software is becoming an integral part of our daily lives. It is omnipresent and plays an important role in every aspect of human life. We can find software in almost all electronic devices around us. Even routine activities such as checking the weather prediction or transportation schedules rely on software. It is not a surprise that software has the highest growth rate in the technological industry [Sta]. We can hardly imagine what our life would be without computer software.

Software development is an error-prone human task. This is because there are many abstractions involved from software to the executable code in hardware. Due to these abstractions, programmers typically specify software's behavior in an informal way. Then, they transform the informal description (through a systematic approach) into a formal programming language. There are many ambiguities in this process that make it hard to see whether the implementation satisfies the specification. Moreover, complexity of software increases rapidly. As software becomes larger in size, it becomes more complicated to understand and analyze its behavior. This is because the interaction of software components grows exponentially. Since our life heavily depends on software, reliability of software becomes a crucial factor in software development. That means, we should guarantee that software behaves correctly in all different scenarios. Hence, software should be functionally correct. Particularly, in critical systems where failures can result in serious damage to human life and environment; see [LT93, Pra95, LLF+96, DC12, GKTZ12, Wit16] for examples of disasters that occurred due to software failures.

In addition to reliability, also increasing the performance of software is demanded by users. As a software user, we desire that software reacts quickly to our requests. This requirement becomes essential in some situations, for instance in critical systems, where incorrect decisions may be made, due to slow response from the software (e.g., during a surgery on a patient). Thus, from the user perspective, software should behave correctly and respond quickly. These two

properties are important factors in software quality. Therefore, it is essential to develop techniques to guarantee the *reliability* of *high-performant* software.

To achieve performance, new hardware and accordingly new programming paradigms are emerging. There was a famous prediction about hardware trends (known as Moore's law), which states that *"the number of transistors on a CPU chip will double every 18 months"*. Similarly, Dennard scaling law states that as the number of transistors get smaller, their power density remains constant. Therefore, CPU manufacturers can raise clock frequency without significantly increasing overall circuit power consumption. This had been true for a long time. However, we observe that the number of transistors does not increase in the scale of Moore's law anymore. Moreover, Dennard scaling law has now ended. Instead of increasing the speed of a uniprocessor, hardware manufacturers produce multi-core processors. That means, there are multiple-cores on a CPU chip. Accordingly, multi-threaded programming became a common programming practice, because multi-threaded programs benefit from the multi-core processors to improve performance.

From sequential programming on uniprocessors to multi-threaded programming on multi-core processors, we are now in the era of massively parallel programming on many-core processors. One of the promising many-core processors is Graphics Processing Units (GPUs). These were initially invented for image rendering purposes. However, GPUs gradually evolved to be used for general purpose applications known as General Purpose GPUs (GPGPUs)[1]. GPGPUs have many more cores which are simpler, but more tuned for data parallelism than CPUs. GPGPUs naturally support parallel execution, with many threads cooperating together, executing the same instructions, but on different data. This paradigm is called Single Instruction Multiple Data (SIMD). CUDA [NBGS08] and OpenCL [Khr08] are two dominant GPU programming languages.

GPU programs improve the performance significantly and are widely-used in industry. However, they provide a risk for the reliability, as GPU programming is not an easy task. Programmers need full knowledge of both GPU hardware and the programming model. Therefore, unintended errors occur more frequently in GPU programming [vdHWvdBH20]. Thus, due to the wide-spread use of GPU programs, providing reliability and correctness of GPU programs is of the utmost importance.

The traditional approach to provide reliability and correctness of software, and GPU programs in particular, is program testing. In this approach, we observe the behavior of programs by running or simulating them using some input (e.g., test cases). Hence, its aim is to decrease the number of bugs as much as possible. This is a straightforward approach to find bugs, but as Dijkstra stated

[1] In this thesis, we use GPU and GPGPU interchangeably.

"program testing can be used to show the presence of bugs; but never to show their absence!" [DDH72]. Therefore, program testing in isolation does not guarantee the correctness of software. In addition, in some cases providing intense testing requires human manpower at high cost.

A more elegant approach to prove the correctness of (GPU) programs is deductive program verification. In this approach, we annotate programs by specifying pre- and postconditions, invariants and intermediate properties written in first-order logic. Then, using a proof system we can prove that: if the preconditions hold in the initial state of a program, after termination of the program, all final states satisfy the postconditions. The advantage of this technique is that we guarantee the correctness of software.

Reasoning about GPU programs using deductive techniques is challenging due to the many different interleavings of arbitrary threads. Moreover, the hierarchical structure of threads and different levels of memory accesses on GPUs makes it more challenging to prove the correctness of GPU programs. Thus, it is essential to develop deductive techniques to prove that first, GPU programs are data race-free. A data race is a situation where two or more threads may access the same memory location simultaneously where at least one of them is a write. Second, we also have to prove that the programs are functionally correct, i.e., they actually produce the result we expect.

Therefore, the goal of this thesis is to introduce a framework to develop *correct optimized GPGPU programs* during the GPU development cycle. Concretely, the aim is to first prove the correctness of unoptimized GPU programs. An unoptimized GPU program is considered as a naive implementation of an algorithm on a GPU where the focus is on its functionality rather than on the performance. Second, to preserve their correctness after applying multiple optimizations.

To accomplish this goal, the first step is to develop and use deductive techniques to reason about unoptimized complicated GPU programs, i.e., GPU implementations of algorithms that are not trivial to reason about. Hence, the first part of this thesis studies deductive techniques to prove data-race freedom and functional correctness of some complicated unoptimized GPU programs and uses them on several case studies. Concretely, we show data-race freedom and functional correctness of GPU-based implementations of prefix sum, stream compaction, summed-area table and Bellman-Ford shortest path algorithms. These are important algorithms on GPUs that are used as building blocks for other algorithms. Moreover, we show how to prove the permutation property of (GPU-based) parallel odd-even transposition sort algorithm. Interestingly, we generalize the permutation proof technique to be used for any parallel and sequential swap-based sorting algorithm, and show how this used to prove the permutation property of some well-known sequential sorting algorithms, namely bubble sort, selection sort, insertion sort, quick sort,

two in-place merge sorts and TimSort.

Proving an unoptimized version of a GPU program is not sufficient for the goal of the thesis, because, the typical GPGPU development cycle is an iterative process. Programmers either design a parallel algorithm from scratch or parallelize an existing sequential algorithm on GPUs. In both cases, they first implement a naive unoptimized version of the program. Then after validating its correctness, programmers manually apply a chain of optimizations step-by-step to benefit from the underlying hardware and to increase the performance of the program. That means, optimizing GPU programs currently requires programmers to have expert knowledge about the hardware. In particular, one does not (yet) rely as much on the compiler applying optimizations as is typically done when developing CPU programs. To make program optimization easier, there has been some work on defining transformations that can be applied to automate the optimization process [RKBA+13, YXKZ10]. However, each optimization might introduce non-trivial errors to the program and little effort has yet been made to guarantee that functional correctness of programs is preserved during such transformations. Particularly, in deductive program verification, when we apply an optimization technique, we should change the annotations accordingly, to reverify the optimized GPU program. However, transforming the annotations for verification manually along with the optimizations would be very time-consuming and error-prone.

Therefore, the second part of the thesis shows how to automatically apply GPU optimizations to an unoptimized verified GPU program to guarantee that the final optimized GPU program is still correct in the development process. Concretely, we introduce *annotation-aware (source-to-source) transformations* such that if an original program is verified, then the program *along with its annotations* are transformed and the resulting annotated optimized program can still be verified again. To demonstrate this technique, we apply it to the case studies that are verified in the first part of the thesis, i.e., we show how to apply several optimizations to the verified programs while preserving their provability. The GPU optimizations that we consider in this thesis are loop unrolling, iteration merging, matrix linearization, data prefetching, tiling and kernel fusion. As a consequence, the approach in this thesis results in correct optimized GPU programs.

Contributions. The main contributions of this thesis are:

1. We show that the parallel GPU-based prefix sum algorithms by Blelloch and Kogge-Stone are data race-free and functionally correct for any arbitrary size of input, using a deductive approach.
2. We show that the parallel GPU-based stream compaction and summed-area table algorithms are data race-free and functionally correct for arbitrary input sizes, using a deductive approach.

3. We show that the parallel GPU-based Bellman–Ford algorithm is data race-free and functionally correct for arbitrary input sizes, using a deductive approach.
4. We propose a generic technique to prove the permutation property of bubble sort, selection sort, insertion sort, parallel odd-even transposition sort, quick sort, two in-place merge sorts and TimSort using deductive approach.
5. We show how to automatically apply well-known GPU optimizations, loop unrolling, iteration merging, matrix linearization, data prefetching, tiling and kernel fusion, to the verified GPU programs while preserving the provability of the programs.

All proofs and implementations in this thesis have been formalized in VerCors. The sources can be found in a repository available at:

`https://doi.org/10.4121/19055453`

We often omit details or proofs from lemmas given in this thesis, as the formalization is quite involved. These details are deferred to the encodings in VerCors.

1.1 Thesis Outline

Chapter 2 provides preliminaries about deductive verification technique and GPU programming. Moreover, it discusses deductive approaches and tools for correctness of GPU programs.

Part I of this thesis is organized as follows:

Chapter 3 proves two parallel prefix sum algorithms on GPUs. Concretely, this chapter proves data-race freedom and functional correctness of Blelloch's and Kogge-Stone's prefix sum algorithms. This chapter is based on the following publications:

- Mohsen Safari, Wytse Oortwijn, Sebastiaan Joosten, and Marieke Huisman. Formal verification of parallel prefix sum. In Ritchie Lee, Susmit Jha, Anastasia Mavridou, and Dimitra Giannakopoulou, editors, *NASA Formal Methods*, pages 170–186, Cham, 2020. Springer International Publishing
- Mohsen Safari and Marieke Huisman. Formal verification of parallel prefix sum and stream compaction algorithms in CUDA. *Theoretical Computer Science*, 912:81–98, 2022

Chapter 4 proves two applications of parallel prefix sum algorithms on GPUs. Concretely, this chapter proves data-race freedom and functional correct-

ness of stream compaction and summed-area table algorithms on top of the two verified prefix sum algorithms. This chapter is based on the following publications:

- Mohsen Safari and Marieke Huisman. Formal verification of parallel stream compaction and summed-area table algorithms. In Violet Ka I Pun, Volker Stolz, and Adenilso Simão, editors, *International Colloquium on Theoretical Aspects of Computing*, Lecture Notes in Computer Science, pages 181–199. Springer, 2020

Chapter 5 proves a parallel Single Source Shortest Path (SSSP) algorithm on GPUs. Concretely, this chapter proves data-race freedom and functional correctness of the parallel Bellman-Ford algorithm. This chapter is based on the following publications:

- Mohsen Safari, Wytse Oortwijn, and Marieke Huisman. Automated verification of the parallel Bellman–Ford algorithm. In Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi, editors, *Static Analysis*, pages 346–358, Cham, 2021. Springer International Publishing

Chapter 6 proposes an generic approach to prove the permutation property of some well-known sorting algorithms. Concretely, this chapter proves the permutation property of bubble sort, selection sort, insertion sort, parallel odd-even transposition sort, quick sort, two in-place merge sorts and TimSort. This chapter is based on the following publications:

- Mohsen Safari and Marieke Huisman. A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In Brijesh Dongol and Elena Troubitsyna, editors, *Integrated Formal Methods*, Lecture Notes in Computer Science, pages 257–275. Springer, 2020

Part II of this thesis is organized as follows:

Chapter 7 introduces the idea of annotation-aware transformation. It illustrates how to apply GPU optimization techniques such as loop unrolling, iteration merging, matrix linearization, data prefetching, tiling and kernel fusion, to the verified GPU programs automatically while preserving the provability of the programs. This chapter is based on the following publication:

- Ömer Şakar, Mohsen Safari, Marieke Huisman, and Anton Wijs. Alpinist: An annotation-aware GPU program optimizer. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and*

Analysis of Systems, pages 332–352, Cham, 2022. Springer International Publishing

Chapter 8 concludes the thesis and discusses future work.

CHAPTER **2**

Background

This chapter provides the necessary preliminary information that is required for the rest of this thesis. First, it explains deductive program verification. Second, it discusses the GPGPU architecture and programming model. Third, it provides an overview of several well-known approaches and tools for reasoning about GPU programs. Finally, this chapter discusses the VerCors program verifier in detail, and how to verify GPU programs using this tool, as we use VerCors in the thesis.

2.1 Deductive Program Verification

Tony Hoare, Robert Floyd and Edsger Dijkstra are the pioneers that developed a technique to deductively reason about sequential programs. Floyd-Hoare established a framework in their seminal work [Flo67, Hoa69] to reason about correctness of programs. The core idea is that a program C is annotated by preconditions $\mathcal{P}$ and postconditions $\mathcal{Q}$. This is shown in the form of (Hoare) triples:

$$\{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$$

where $\mathcal{P}$ and $\mathcal{Q}$ are written as logical assertions, typically in first-order logic. The semantics of Hoare triples is that whenever $\mathcal{P}$ holds in the initial state before the execution of C, then $\mathcal{Q}$ will hold afterwards, or C does not terminate. As we do not consider the termination problem, this is called partial correctness.

Hoare logic provides axioms and inference rules that allow us to check whether a program annotated with Hoare triple annotations is correct. Floyd-Hoare only proposed a method to check a proof, but did not show how to generate such a proof. Dijkstra showed a computational approach [Dij76] (which thus can be mechanized) to generate such a proof using predicate transformers. In this method, an annotated program can be transformed into verification conditions.

```
1  precondition {a == A, b == B}
2  void swap(int a, int b){
3    {b == B ∧ a == A}
4    int temp = a;
5    {b == B ∧ temp == A}
6    a = b;
7    {a == B ∧ temp == A}
8    b = temp;
9    {a == B ∧ b == A}
10 }
11 postcondition {a == B ∧ b == A}
```

Figure 2.1: An example of deductive verification approach.

A verification condition is a logical formula whose validity implies the correctness of the program. As a result, verification conditions can be discharged by SAT solvers [CKL04, CKSY05, IYG+08] or SMT solvers [LQ08, BMR12, Sch16b]. Several tools have been developed as program verifiers that combine and automate this process; e.g., Why3 [FP13], Frama-C [KKP+15], Boogie [BCD+05], Viper [JKM+14, MSS16] and VerCors [BHM14].

Example Figure 2.1 shows how to reason about programs deductively. The program is swapping the values of two variables `a` and `b` (lines 4, 6 and 8)[1]. The precondition specifies the initial values of `a` and `b` before executing the function (line 1). The postcondition specifies that the values of `a` and `b` are swapped after termination of the function (line 11). To prove correctness of the program using deductive verification, we follow a backward approach. That means, we start from the postcondition and by applying several axiomatic rules according to the program statements, we can conclude whether the precondition holds or not.

In this example, from the postcondition (line 9) and the statement in line 8, we replace `b` by `temp` (line 7). Then, according to the statement in line 6, we replace `a` by `b` (line 5). Finally, according to the statement in line 4, we replace `temp` by `a` (line 3). As the precondition logically implies the property in line 3, we conclude that the swap program is functionally correct. □

Separation logic [Rey02] was originally proposed as an extension of Hoare logic to reason about heap-manipulating programs. In contrast to Hoare logic, assertion properties in states are separated over store and heap locations. One of the important rules in separation logic is the *frame rule*:

[1] We assume the function will be called by reference. Therefore, the swap is visible from caller.

$$\frac{\{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \qquad (Mod(C) \cap FV(\mathcal{R}) = \emptyset)}{\{\mathcal{P}{*}\mathcal{R}\}\, C\, \{\mathcal{Q}{*}\mathcal{R}\}} \quad \big[\textbf{Frame Rule}\big]$$

The frame rule indicates that if $\{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ holds in a local state, we can conclude that $\{\mathcal{P} * \mathcal{R}\}\, C\, \{\mathcal{Q} * \mathcal{R}\}$ also holds for an extended state. The predicate $\mathcal{P} * \mathcal{R}$ asserts that $\mathcal{P}$ and $\mathcal{R}$ hold on separate disjoint memory locations. This is shown by the separating conjunction operator $*$. The side condition specifies that C does not modify any free variables in $\mathcal{R}$.

The frame rule in separation logic makes it also suitable to reason about multi-threaded programs [O'H07]: different threads that work on disjoint parts of the heap do not interfere and therefore can be verified in isolation. This is formulated in the *parallel rule*:

$$\frac{\{\mathcal{P}_1\}\, C_1\, \{\mathcal{Q}_1\} ... \{\mathcal{P}_n\}\, C_n\, \{\mathcal{Q}_n\} \qquad (S_1 \ and \ S_2)}{\{\mathcal{P}_1 {*} ... {*} \mathcal{P}_n\}\, C_1 || ... || C_n\, \{\mathcal{Q}_1 {*} ... {*} \mathcal{Q}_n\}} \quad \big[\textbf{Parallel Rule}\big]$$

The parallel rule indicates that if $\mathcal{P}_1$ to $\mathcal{P}_n$ hold in n disjoint memory locations, and each thread operates on one partition, then we can reason about each thread in isolation to conclude that $\mathcal{Q}_1$ to $\mathcal{Q}_n$ hold. Side condition S_1 states that a thread cannot change variables of other threads, and side condition S_2 states that a thread executing C_i should not modify free variables in P_j or Q_j $(i \neq j)$.

Separation logic enables a modular way to reason about concurrent programs. However, separation logic is not able to allow multiple threads to have simultaneous read accesses to a shared location. To solve this, Bornat et al. [BCOP05] propose to use fractional permissions [Boy03] and add permission accounting to separation logic. A full permission 1 denotes a write permission, whereas any fraction in the interval (0, 1) denotes a read permission. A write permission can be split into multiple read permissions and read permissions can be added up, and transformed into a write permission if they add up to 1. The soundness of the logic ensures that for each memory location, the total number of permissions among all threads does not exceed 1. In this way, data-race freedom of multi-threaded programs can be proven, because for one shared location either all threads have read permissions, or only one thread has write permission. Therefore read and write inconsistency never happens.

2.2 GPGPUs

Architecture Figure 2.2 shows an overall view of the GPU architecture and its programming model[2]. GPGPUs have many simpler, but more efficient cores than

[2]The figure is taken from NVIDIA CUDA C Programming Guide: `http://docs.nvidia.com/cuda/-cuda-c-programming-guide/index.html`.

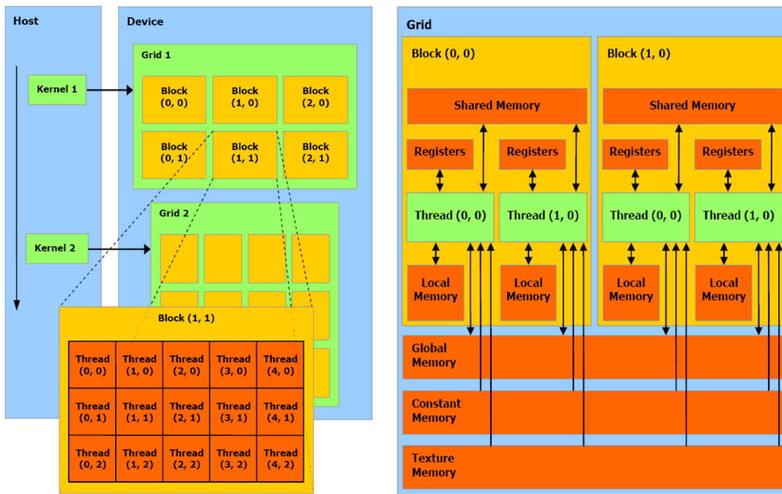

Figure 2.2: GPGPU architecture and programming model

CPUs that can run many threads simultaneously. A GPGPU has several Multi-Processors (MPs) and each MP has some Streaming Processors (SPs). Threads are organized into blocks and the scheduler of the GPU assigns the thread blocks to MultiProcessors. Then the threads within one block can use the Streaming Processors available on that MultiProcessor to execute instructions in parallel. One thread block can only occupy one MultiProcessor, but multiple blocks can be assigned to one MultiProcessor.

GPGPUs have a hierarchical memory structure. The fastest memory is a register, which is local to a single thread. Each MultiProcessor has a shared memory, which is the second-fastest memory. Threads from one thread block cannot access the shared memory of other blocks. Finally, the slowest memory is global memory. This is accessible by all threads and by the CPU and thus can be used for communication between all threads and from and to the CPU. Moreover, there are two disjoint types of read-only memory on GPUs: constant and texture memory. The capacity of read-only memory is limited.

Programming Model CUDA and OpenCL are the two most dominant GPU programming models. CUDA is an extension of C and it allows software developers to implement parallel algorithms on NVIDIA GPUs. A CUDA program is a CPU-GPU program. It means part of the program runs on the CPU and the other part runs on the GPU. The CPU part is called *host* and the GPU part is called *kernel*. The typical workflow in a CUDA program is that we copy data from the host

CUDA	OpenCL
grid	computation domain
thread block	work-group
thread	work-item
shared memory	local memory
local memory (register)	private memory
__syncthreads()	barrier()

Figure 2.3: CUDA vs. OpenCL terminology.

to the *device* (GPU), and then we call the kernel from the host, running on the device. Each thread executes the kernel code on its own part of the data. Finally, we copy data back from the device to the host.

In the CUDA programming model, programmers can define threads in a hierarchical level as grids and blocks. A block indicates a number of threads running on one MP. A grid shows the total number of blocks to run on one GPU. Both grids and blocks can be defined in one, two or three dimensions by the programmer. CUDA provides a mechanism for barrier synchronization amongst the threads within a block. That means, when threads (within a block) hit a barrier, they wait for all other threads (of the same block) to hit the same syntactical barrier. Then, after that, they resume their execution. Note that there is no programming primitive for inter-block synchronization. Thus, inter-block synchronization can only be achieved using the host side. That means, we should split the kernel into two different kernels and invoke them in sequence from the host. There are also primitive atomic operations which can be used to avoid read/write inconsistencies in global and shared memories.

OpenCL is an open-source standard to write parallel programs on GPUs which is developed by the Khronos group [Khr08]. The programming model is very similar to CUDA. However, in contrast to CUDA, OpenCL programs can be executed on different GPU architectures, e.g., NVIDIA, ARM, Intel. Moreover, the terminology of concepts varies in CUDA and OpenCL. For instance, the built-in synchronization function in CUDA is called `__syncthreads()` whereas in OpenCL it is called `barrier()`. Figure 2.3 shows some different terminologies of the same concepts that are used in CUDA and OpenCL. In this thesis, we use the CUDA terminology.

Example Figure 2.4 shows a CUDA (left) and OpenCL (right) kernel that rotates the elements of an array to the left. We assume there is only one thread block. In both programs, each thread (*tid*) stores its right neighbor in a tempo-

```
1  __global__ void CUDA_LR(int *array,
      int size) {
2    int tid = threadIdx.x;
3    int temp;
4
5    if(tid != size-1){
6      temp = array[tid+1];
7    }else{
8      temp = array[0];
9    }
10
11   __syncthreads();
12
13   array[tid] = temp;
14 }
```

```
1  __kernel void OpenCL_LR(int *
      array, int size) {
2    int tid = get_local_id(0);
3    int temp;
4
5    if(tid != size-1){
6      temp = array[tid+1];
7    }else{
8      temp = array[0];
9    }
10
11   barrier(CLK_LOCAL_MEM_FENCE);
12
13   array[tid] = temp;
14 }
```

Figure 2.4: A simple CUDA (left) and OpenCL (right) kernel.

rary location (i.e., *temp*), except thread *size*-1 which stores the first element in the array (lines 5-9). The keywords `threadIdx.x` and `get_local_id(0)` are used to get thread identifiers in a block in CUDA and OpenCL, respectively. Then each thread synchronizes in a barrier (line 11). After that, each thread writes the value read into its own location at index *tid* in the array (line 13). In OpenCL, the barrier function has two arguments: `CLK_LOCAL_MEM_FENCE` and `CLK_GLOBAL_MEM_FENCE`. They are explicitly used to flush local and global memories. In contrast, the synchronization function in CUDA is used without any parameters for flushing both shared and global memories. Note that the keywords `__global__` and `__kernel` indicates that these functions are executed on GPUs. □

2.3 GPGPU Verification Techniques

There are three main issues related to correctness of GPU programs: data races, barrier divergence and functional correctness. A *data race* is a situation where two or more threads may access the same memory location simultaneously and at least one of them is a write. A *barrier divergence* happens when threads from the same thread block diverge and hit different (syntactical) barriers. In these two cases the behavior of the GPU program is undefined. *Functional correctness* captures whether the program behaves as we expect. There are some techniques and tools developed to reason about GPGPU programs. The aim of most of the tools is to detect data races in GPU programs. Some of them also support barrier-divergence detection. There are only a few tools that can establish functional correctness of GPU programs, which is the most difficult part in verification. We can categorize

these approaches and tools as dynamic, hybrid and static. We briefly introduce some of them, and go deeply into the VerCors verifier as this is the tool that we use in this thesis.

Dynamic Tools that use dynamic approaches are CUDA-MEMCHECK [Nvi19] and Oclgrind [PMS15]. These tools instrument the programs and dynamically record memory accesses. Then they check the logs to detect memory races. CUDA-MEMCHECK is developed by NVIDIA to detect and debug memory errors in CUDA programs. Moreover, it detects out-of-bounds and misaligned memory accesses. Oclgrind is a simulator to analyze and debug OpenCL programs. It provides memory access checking and data race detection of OpenCL kernels. Data races that these tools detect are true bugs (no false positive) as they run or simulate the programs. However, the disadvantage of these tools are that they do not guarantee the absence of data-races.

Hybrid There are some hybrid approaches that combine static and dynamic approaches to take advantages of the two approaches. GRace [ZRQA11] and GM-Race [ZRQA13] are developed to detect data-races in CUDA programs as a hybrid tool. These tools first statically detect, or prune possible data races for most memory accesses. This is because most GPU memory access patterns such as a[tid] = 5 can be determined at compile time. Then, for the remaining memory accesses, they use a dynamic approach to instrument programs to detect data races at runtime. SESA [LLG14] is another tool that benefits from both static and dynamic approaches to race-check CUDA programs. Another hybrid technique is a dynamic symbolic approach known as concolic. In this technique, program variables are considered as symbolic variables. According to concrete values, a specific path of the program is covered. Concurrently a symbolic execution is run (over symbolic variables) to collect a set of symbolic constraints known as path conditions. Each path condition indicates how a set of concrete values could execute different execution path in the program. By negating the last condition in the path each time, a constraint solver can generate a new concrete input to uncover more paths in the program. The aim is to generate test cases with high path coverage. This approach is implemented in GKLEE [LLS+12] and KLEE-CL [CCK11]. GKLEE is targeting CUDA and KLEE-CL is targeting OpenCL programs. The drawback of this approach is poor scalability due to the exponential path growth. Moreover, constraint solvers incur high overhead during this approach. CLFuzz [PR20] is proposed as an improvement over this approach by combining mutation-based fuzzy testing and symbolic constraint solving. CLFuzz uses fuzzy testing to generate test cases, and if it fails to uncover new paths, it switches to symbolic constraint solving. CLFuzz is applied to OpenCL programs only.

Static Some other tools are developed based on static approaches to reason about GPU programs. The advantage of static approaches is that they guarantee the absence of bugs. The disadvantage of this approach is that we have to provide problem-specific invariants. The invariant is the key in the proof of static approaches. Some tools provide (semi-)automatic invariant inference of programs, but in general this is a hard problem. Therefore, investigation of invariants requires human effort. PUG [LG10] is a static verifier that verifies data-race freedom and barrier-divergence freedom of CUDA programs. A CUDA program is translated into verification conditions and then using an SMT-based solver, it proves data-race freedom of the program. The advantage of PUG is that it infers loop invariants using loop abstraction techniques. However the user can also specify invariants manually. The unsoundness problem arises that PUG assumes these (automatically inserted or user-supplied) invariants are true without being checked.

GPUVerify [BCD+15] verifies data-race freedom and barrier-divergence freedom in both CUDA and OpenCL kernels. Instead of translating the kernel into logical formulae, GPUVerify translates the kernel into a sequential program. Then it leverages verifiers that reason about sequential programs (e.g., Boogie). It uses the idea of two-thread abstraction: if a kernel is proven data-race free and barrier-divergence free for a pair of distinct but otherwise arbitrary threads, then the correctness of the kernel can be concluded. They provide a pen-and-paper proof of the two-thread reduction. In addition, GPUVerify provides loop invariants while transforming the program. In contrast to PUG, these invariants are checked and cannot lead to false negatives. Faial [CLRZ21] is similar to PUG, but it supports inter-thread data races in synchronized loops. Moreover, it provides the first machine-checked proof of the two-thread abstraction idea.

The only tools that can reason about functional correctness of GPU programs are VeriCUDA [KI13, KI17] and VerCors [BHM14]. VeriCUDA extends Hoare logic to prove data-race freedom and functional correctness of CUDA programs. It also takes advantage of two-thread reduction to reason about data-race freedom. VerCors benefits from permission-based separation logic to prove data-race freedom and functional correctness of of GPU programs. In both approaches users should specify pre- and postconditions and invariants manually into the program. In contrast to VeriCUDA, VerCors provides more evidence of its usability by verifying a broad range of examples. This is observable by comparing the GitHub page of VeriCUDA in [Verb] and VerCors in [Vera]. Because of that, we use VerCors as a practical tool to automate reasoning, and prove data-race freedom and functional correctness of GPU programs in this thesis. Hence, the next subsection provides a more in depth explanation of VerCors.

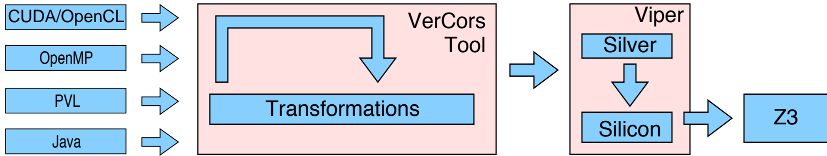

Figure 2.5: VerCors tool set overall architecture

2.3.1 VerCors

VerCors[3] is a program verifier to specify and verify concurrent and parallel programs. Figure 2.5 shows the architecture of the VerCors verifier. It supports high-level languages such as (subsets of) Java, CUDA, OpenCL, OpenMP and PVL. PVL is VerCors' own language for prototyping new features. VerCors can be used to verify data-race freedom and functional correctness of (concurrent) programs. VerCors requires programs to be annotated with pre/post-conditions using permission-based separation logic [BCO05, BCOP05, Boy03]. The annotated program is transformed in several steps into the input language of Viper (named Silver) [JKM+14, MSS16]. Then the backend of Viper (Silicon) translates the annotated program into proof obligations, which are discharged to an automated theorem prover; in our case Z3 [DMB08]. In particular, VerCors reads the input file and transforms it into an internal AST (written in the so-called Common Object Language or COL). All of the supported input languages can be transformed into COL. VerCors applies several transformations on the COL AST. These transformations change the COL AST to a Silver AST.

Example Figure 2.6 shows how we encode and verify data-race freedom and functional correctness of the same GPU program as in Figure 2.4 using VerCors. To verify GPU programs in VerCors, the algorithms are encoded into the PVL language. PVL defines several parallel programming constructs (i.e., parallel blocks, barriers, atomics, etc) that we use to encode GPU programs.

The figure contains a function named *leftRotation* which is considered as the host (CPU) part. Inside the function, there is a parallel block named *kernel* (lines 12-23). The keyword `par` is used to define the parallel block, followed by an arbitrary name for the block. The par block first specifies the number of threads in the parallel block, as well as a name for the thread identifier. In this example, we have *size* threads in the range from 0 to *size*-1 and *tid* is used to refer to each thread (line 12). The body of the parallel block is executed by each thread as the GPU

[3]See `https://utwente.nl/vercors`.

```
1  /*@ context_everywhere array != NULL && array.length == size;
2      requires (\forall* int i; i ≥ 0 && i < size; Perm(array[i], 1));
3      ensures (\forall* int i; i ≥ 0 && i < size; Perm(array[i], 1));
4      ensures (\forall int i; i ≥ 0 && i < size; i != size-1 ?
5                array[i] == \old(array[i+1]) : array[i] == \old(array[0])); @*/
6  void leftRotation(int[] array, int size){
7    /*@ requires tid != size-1 ? Perm(array[tid+1], 1\2) : Perm(array[0], 1\2);
8        requires Perm(array[tid], 1\2);
9        ensures Perm(array[tid], 1);
10       ensures tid != size-1 ==> array[tid] == \old(array[tid+1]);
11       ensures tid == size-1 ==> array[tid] == \old(array[0]); @*/
12   par kernel(int tid = 0..size){
13     int temp;
14     if (tid != size-1) { temp = array[tid+1]; }
15     else { temp = array[0]; }
16
17     /*@ requires tid != size-1 ? Perm(array[tid+1], 1\2) : Perm(array[0], 1\2) **
18                 Perm(array[tid], 1\2);
19         ensures Perm(array[tid], 1); @*/
20     barrier(kernel);
21
22     array[tid] = temp;
23   }
24 }
```

Figure 2.6: Verification of the left rotation program (Figure 2.4) in VerCors.

part. The keyword **barrier** and the name of the parallel block as an argument (e.g., *kernel* in the example) are used to define a barrier in PVL (line 20).

To verify this function in VerCors, we annotate the barrier, in addition to the function and the parallel block. To specify permissions, we use predicate `Perm(L, π)` where `L` is a heap location and π a fractional value in the interval $(0, 1]$[4]. Pre- and postconditions, (denoted by keywords **requires** and **ensures**[5], respectively in lines 2-5, 7-11), must hold at the beginning and the end of the function (or par block), respectively. The keyword `context_everywhere` is used to specify an invariant (line 1) that must hold throughout the function (including the par block). As precondition of the function, we have write permission over all locations in the array (line 2). At the beginning of the parallel block, each thread reads from its right neighbor, except thread *size*-1 which reads from location 0 (lines 14-15). Therefore, we specify read permissions as precondition of the parallel block in line 7. Since after the barrier each thread (*tid*) writes into its own location at index *tid*,

[4]The keywords **read** and **write** can also be used instead of fractions in VerCors.
[5]For the presentation purpose, in the rest of this thesis we use **req** and **ens** in the annotations.

we change the permissions in the barrier such that each thread has write permissions to its own location (lines 17-19). When a thread reaches the barrier, it has to fulfill the barrier preconditions, and then it may assume the barrier postconditions. Moreover, the barrier postconditions must follow from the barrier preconditions. Therefore, each thread has initially read permission in its own location as well (line 8). As a result, the accumulation of available permissions in each location of the array is 1 and we can change it into write permission in the barrier.

As postcondition of the parallel block (1) first each thread has write permission to its own location (this comes from the postcondition of the barrier) in line 9 and (2) the elements are truly shifted to the left (lines 10-11). From the postcondition of the parallel block, we can establish the corresponding postcondition for the function (lines 3-5). Note that the keyword `\old` is used for an expression to refer to the value of that expression before entering a function (lines 4-5). Moreover, `\forall*` indicates universal separating conjunction over permission predicates and `\forall` denotes standard universal conjunction over logical predicates. In addition, we use && for logical conjunction (line 1) and ** as separating conjunction in separation logic (lines 17-18). □

Part I

Correct GPU Programs; case studies

CHAPTER **3**

Verification of Parallel Prefix Sums

Prefix sum is a frequently used building block operation on GPUs. In this chapter, we use deductive program verification based on permission-based separation logic, as supported by VerCors, to show correctness of the two most frequently used *parallel* in-place prefix sum algorithms, namely Blelloch's and Kogge-Stone's algorithms, for an *arbitrary array size*. Interestingly, the correctness proof for the second algorithm reuses the auxiliary lemmas that we needed to create the first proof.

3.1 Introduction

One of the algorithms for which several parallel (GPU-based) implementations have been proposed is the prefix sum algorithm [BK82, KS73, Ble93, Skl60, SLO06]. It takes an array of integers and, for each element, it computes the sum of the previous elements. The prefix sum algorithm is used in many other algorithms, e.g. in radix sort, quick sort, to solve recurrences, and in tridiagonal linear systems; see Blelloch [Ble93]. For this reason, prefix sum is an interesting case study from a verification point of view.

Blelloch introduced a parallel in-place prefix sum algorithm and Harris [HSO07] adapted it for GPUs. Kogge-Stone [KS73] proposed a different parallel in-place prefix sum algorithm and Horn [Hor05] adapted it for GPUs. These two parallel versions [Ble93, KS73] are the most used in practice and are available as a primitive operation in many libraries (e.g., AMD APP SDK[1], NVIDIA CUDA SDK[2]).

The GPU-based implementations of these two algorithms are widely used, even

[1] `http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk`

[2] `https://developer.nvidia.com/gpu-computing-sdk`

as a building block for other algorithms (e.g., sorting). Therefore, the correctness of these algorithms is of utmost importance. This means that the algorithms must be memory and thread safe (i.e. free of data races), and that they must be functionally correct, i.e. it actually produces the result we expect. Concretely, in this case functional correctness means that the result must be the prefix sum of the input. In general, proving functional correctness of parallel programs is a difficult task. In particular, proving the functional correctness of these two parallel prefix sum algorithms is challenging for several reasons. First, both algorithms are in-place, i.e. we need to reason about values that are unstable and change during the algorithm. Second, the computational pattern of the algorithms makes it complex to reason about the final result. Therefore, it is a challenge to find suitable properties to relate the internal computation steps in the algorithms to the final result. In particular, in Blelloch's algorithm, there are two independent, but closely related phases with different computation pattern in each phase, which makes the verification harder. As a result, establishing functional correctness of the two algorithms is non-trivial.

For the verification, we use deductive verification, as supported by VerCors to prove data-race freedom and functional correctness of two parallel prefix sum algorithms. First, we show how to verify the correctness of Blelloch's algorithm. An important feature of our verification is that it is a non-trivial example of how ghost code helps to reason about in-place algorithms. Ghost code is not part of the algorithm and is used purely for verification purposes; see [FGP16]. Second, we show how we can verify a different parallel in-place prefix sum algorithm, Kogge-Stone, using the same approach as the first verification. This demonstrates that the verification setup introduced in this chapter (approach, operations and lemma) is not specific to this particular case study and can be used in other verifications. We use PVL that models GPU programming (as discussed in the previous chapter) to encode both algorithms in VerCors and prove their correctness. Third, we show how we add CUDA support in VerCors. Then we discuss CUDA verification challenges of the two algorithms by transforming CUDA programs into the parallel programming constructs of PVL.

As a consequence, our verification in this chapter enables the verification of other complicated parallel algorithms, such as stream compaction and summed-area table algorithms, that are built on top of the prefix sum algorithms (see the next chapter). In the rest of this chapter, we first explain the prefix sum problem. Then we show how to verify Blelloch's algorithm as a solution to the prefix sum problem. Furthermore, we benefit from the first verification to show the correctness of Kogge-Stone's algorithm as another solution to the prefix sum problem.

3.2 Prefix Sum Problem

Given an array of integers, the prefix sum of the array is another array with the same size such that each element is the sum of all previous elements. The prefix sum problem is to find an algorithm such that it satisfies the following:

- INPUT: An array *Input* of integers of size N.
- OUTPUT: An array *Output* of size N such that $Output[i] = \sum_{t=0}^{i} Input[t]$ for $0 \leq i < N$.

In the exclusive prefix sum algorithm, where the ith element is excluded from the summation, the output will be:

- OUTPUT: An array *Output* of size N such that $Output[i] = \sum_{t=0}^{i-1} Input[t]$ for $0 \leq i < N$.

Blelloch [Ble93] introduced an *exclusive parallel in-place* algorithm that solves the prefix sum problem. Kogge-Stone [KS73] proposed an *inclusive parallel in-place prefix sum* algorithm. These two parallel versions are frequently used in practice.

3.3 Verification of Blelloch's Parallel Algorithm

In this section, first we explain Blelloch's parallel prefix sum algorithm. Next, we explain how we prove data-race freedom and then functional correctness of this algorithm.

3.3.1 Blelloch's Prefix Sum

Blelloch's algorithm [Ble93] consists of two phases: up-sweep and down-sweep. Figure 3.1 illustrates both up and down-sweep phases visually, and Algorithm 3.1 shows the encoding of the in-place algorithm in VerCors. The up-sweep part in the figure corresponds to lines 2-8 of the algorithm and the down-sweep part corresponds to lines 12-23. Therefore, each iteration in the up/down phases in Algorithm 3.1 (lines 3-8/16-23) correspond to different levels in Figure 3.1. We suppose that at the beginning of Algorithm 3.1, the input and output array have the same values. There is a variable, *stride*, which initially is 1 (line 2) and it is updated in both phases (lines 8 and 23). In the figure, the input values are at level 0 in the up-sweep phase. As we can see, in each iteration of the up-sweep, each pair is summed up at each level. As a result, the last element at the highest level is the summation of the input values. In the down-sweep phase, we first set the last element to 0. Then, we use the partial sums calculated from the up-sweep to

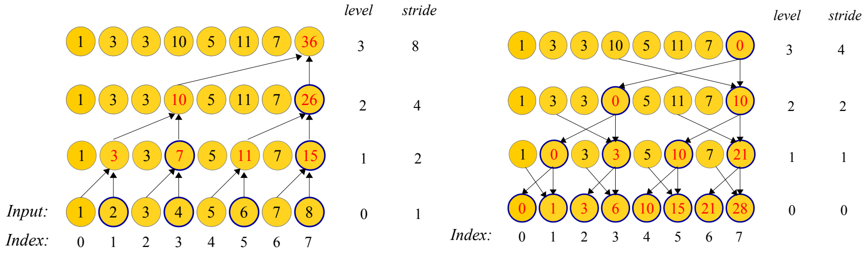

Figure 3.1: After the up-sweep phase (left) and the down-sweep phase (right) in Blelloch's algorithm (two arrows coming to a circle indicates summation and one arrow indicates replacement, red color values show the effect of computations and circles with thick border are *indicators* as in Algorithm 3.1).

Algorithm 3.1 Blelloch's Prefix Sum Algorithm

1: **function** EXCLUSIVE_PREFIXSUM(**int**[] *Input*, **int**[] *Output*, **int** *tid*, **int** N)
2: **int** $indicator = 2 \times tid + 1$; **int** $stride = 1$;
3: **while** $stride < N$ **do**
4: **if** $indicator < N$ && $indicator \geq stride$ **then**
5: $Output[indicator] = Output[indicator] + Output[indicator - stride]$;
6: **Barrier**(tid);
7: $indicator = 2 \times indicator + 1$;
8: $stride = 2 \times stride$;
9:
10: **Barrier**(tid);
11:
12: $indicator = N \times tid + N - 1$; $stride = N/2$;
13: **int** *temporary*;
14: **if** $indicator < N$ **then**
15: $Output[indicator] = 0$;
16: **while** $stride \geq 1$ **do**
17: **if** $indicator < N$ && $indicator \geq stride$ **then**
18: $temporary = Output[indicator]$;
19: $Output[indicator] = Output[indicator] + Output[indicator - stride]$;
20: $Output[indicator - stride] = temporary$;
21: **Barrier**(tid);
22: $indicator = (indicator - 1)/2$;
23: $stride = stride/2$;

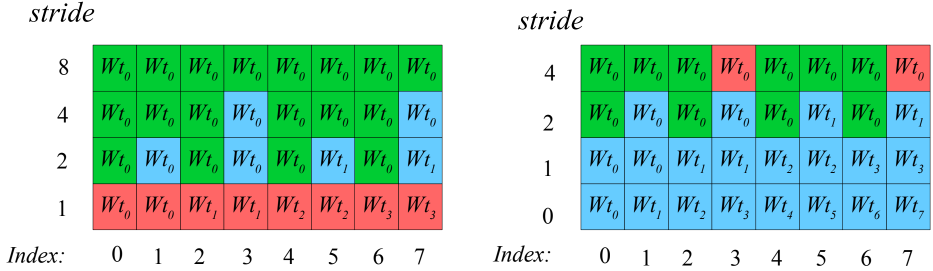

Figure 3.2: Permission patterns for array of length 8: (left) up-sweep and (right) down sweep phases of Blelloch's algorithm (W_{t_i} indicates thread i has write permission, red color indicates initial permissions of active threads, blue shows changes in permission pattern and green shows lost permissions which assigned to thread 0.

compute the prefix sum of the input as indicated at the lowest level in down-sweep. Note that in order to synchronize threads at each level of both phases, a barrier is needed (lines 6 and 21). There is also a barrier between up-sweep and down sweep (line 10). The main purpose of having this barrier is for a specification to redistribute the threads permissions.

3.3.2 Data-Race Freedom

To show that the algorithm is data-race free, we need to specify permissions over resources that are shared among threads. Algorithm 3.1 has two arrays for input and output. Thus, we specify how threads can read or write from these two arrays. In the input array, each thread (tid) only needs read access to location tid. The situation is more complicated for the output array. Figure 3.2 visualizes the permission scheme of threads for the output array graphically. The red elements indicate the initial permissions for both phases. In the up-sweep, each thread needs write access to $indicator$ and $indicator$-$stride$ (line 5 in Algorithm 3.1). Since initially, $indicator$ and $stride$ are $2\times tid+1$ and 1, respectively, we specify write access for each thread to locations $2\times tid+1$ and $2\times tid$, indicated by red color in Figure 3.2 (left). Then, in each iteration, $indicator$ and $stride$ are updated. Therefore, in the barrier of up-sweep (line 6), we change the permissions according to the new values of $indicator$ and $stride$, as shown in blue.

Note that, in each iteration some threads lose permissions, since $indicator$ exceeds the array length (N). According to this scheme, at the end of up-sweep, no threads have permissions left to access elements of the output array due to $indicator >$

N (blue color disappears). However, we need the same pattern of permissions in down-sweep, and in the barrier between up and down sweep (line 10), we cannot invent permissions, but we can only redistribute the current permissions. To solve this, we specify that one arbitrary thread (e.g., thread 0) collects the lost permissions in each iteration (indicated by green). As we can see, at the end of up-sweep, thread 0 has write permission to all locations in the array.

In the down-sweep phase, Figure 3.2 (right), we have the same permission pattern in reverse direction. In down-sweep, thread 0 is the only one whose *indicator* initially is in the bound of the output size (i.e, *indicator* is $N \times tid + N$-1). Thus, initially, thread 0 has write access to *indicator* and *indicator-stride* (indicated in red). Note that, at the beginning of this phase we update *stride* to $N/2$. Thread 0 also has write permission for the rest of elements (indicated by green color), since we need the permissions to redistribute them in the barrier of down-sweep (line 21). As we can see, when we move down, the permission scheme changes according to *indicator* and *stride*. In the end, each thread (*tid*) has write permission to its own location (*tid*) of the output array. In this way threads can safely compute the prefix sum in parallel.

3.3.3 Functional Correctness

To verify functional correctness, we show that at the end of this algorithm, the output array contains the prefix sum of the input array. Proving functional correctness of this algorithm is particularly challenging because:

1. The algorithm is in-place; which means the elements change in each iteration.
2. There are two phases in the algorithm, each with different computations.
3. The intermediate steps are non-trivial, and non-trivial invariants have to be proven to conclude that indeed the prefix sum is proven.

To overcome the above challenges, we keep track of the values in each iteration of the algorithm. For this history of values, we use ghost variables (i.e., for each iteration in both phases, we assign the current values of the output array to a ghost variable of type sequence). Sequences are ordered, finite and immutable collections. There are several advantages of using ghost variables as sequences: (1) it is not required to define permissions over sequences; (2) we can define pure functions over sequences to mimic the actual computations over concrete variables; (3) we can easily prove desired properties (e.g., functional properties) over ghost sequences; and (4) ghost variables can act as histories for concrete variables whose values might change during the program. This gives us a global view (of program states) of how the concrete variables change to their final value.

Moreover, we need to specify invariants that relate the computations in up-sweep

and down-sweep. If we look at the only values that change in Figure 3.1 (red-colored values), we notice that in up-sweep (left) the sum of those values equals the sum of the values in the input array in each iteration. Further, in the down-sweep (right), the red values at each level are the prefix sum of the red values at the corresponding level in the up-sweep. Therefore, our general strategy to tackle the above challenges is:

1. Define different ghost variables in both up-sweep and down-sweep to keep a history of values.
2. Define mathematical functions to update the ghost variables (according to actual computations) in each iteration of the algorithm.
3. Prove functional correctness over the ghost variables using two invariants:
   (a) In up-sweep, the sum of values that change in each iteration equals the sum of the values in the input array.
   (b) In down-sweep, the values that change at each level are the prefix sum of the values that change at the corresponding level in up-sweep.
4. Relate the ghost variables to the actual arrays; i.e., prove that the elements in the ghost variables capture the same elements as in the actual arrays.

Up-Sweep Ghost Variables We go through the steps above to show functional correctness of the algorithm. First, in the up-sweep phase, we define two ghost variables: one to keep track of all values in each iteration as a full history (f_hist with type sequence of sequences), and one to keep history of the only values that change as a partial history (p_hist with type sequence of sequences). We define two different ghost variables because p_hist is used to show preservation of the above two invariants, while f_hist is used to prove that the ghost variable in down-sweep is capturing the elements in the output array. Initially, these two ghost variables contain the values in the input array.

The next step is to define mathematical functions over these ghost variables to update them in the same way as the actual computations do over the actual arrays. To update f_hist in each iteration of up-sweep, we must add a new sequence of current values in the output array to the chain of sequences in f_hist. Therefore, we define a *Build_full_history* function as shown in Figure 3.3. The function takes the previous level in f_hist, named as $f_hist_prev_lvl$, the *stride* and an integer i. The integer i, starts from 0 and increases up to the length of $f_hist_prev_lvl$, indicates the location of elements in $f_hist_prev_lvl$ to be updated. The *Build_full_history* function goes through all elements and updates the elements if the condition $(i\%(2{\times}stride)) = (2{\times}stride\text{-}1)$ && $(i \geq stride)$ holds (lines 11-13), otherwise it keeps the elements unchanged (lines 14-15). Note that, this is a recursive function that captures the same computation as in the algorithm, but over the ghost variable. The postconditions (lines 2-8) specify that

```
1  /*@ req stride > 0 && stride < |f_hist_prev_lvl|;
2      ens |\result| == |f_hist_prev_lvl| - i;
3      ens (\forall int j; j ≥ 0 && j < |\result|; ((i <|f_hist_prev_lvl|) &&
4               ((i+j)≥stride) && (((i+j)%(2×stride)) == (2×stride-1))) ==>
5               \result[j] == f_hist_prev_lvl[i+j] + f_hist_prev_lvl[i+j-stride]);
6      ens (\forall int j; j ≥ 0 && j < |\result|; ((i <|f_hist_prev_lvl|) &&
7               (((i+j)<stride) || (((i+j)%(2×stride)) != (2×stride-1)))) ==>
8               \result[j] == f_hist_prev_lvl[i+j]); @*/
9  pure seq<int> Build_full_history(seq<int> f_hist_prev_lvl, int stride, int i) =
10     i < |f_hist_prev_lvl| ? (
11         ((i%(2×stride)) == (2×stride-1) && (i ≥ stride) ?
12           seq<int> {f_hist_prev_lvl[i] + f_hist_prev_lvl[i-stride]} +
13                    Build_full_history(f_hist_prev_lvl, stride, i+1) :
14           seq<int> {f_hist_prev_lvl[i]} +
15                    Build_full_history(f_hist_prev_lvl, stride, i+1) )) : seq<int> {};
```

Figure 3.3: The *Build_full_history* function.

```
1  //@ req |p_hist_prev_lvl| ≥ 0;
2  pure seq<int> Build_partial_history(seq<int> p_hist_prev_lvl) =
3      1 < |p_hist_prev_lvl| ?
4        seq<int> {head(p_hist_prev_lvl) + head(tail(p_hist_prev_lvl))} +
5        Build_partial_history(tail(tail(p_hist_prev_lvl))) : p_hist_prev_lvl;
```

Figure 3.4: The *Build_partial_history* function.

the result is either the sum of two elements (according to *stride*) if the condition holds (lines 3-5) or unchanged (lines 6-8) otherwise. By applying this function (to *f_hist_prev_lvl*), in each iteration of the algorithm, a full history of values is created like a matrix as sequence of sequences (Figure 3.5 (left)). In the figure, the underlined elements show the locations where the condition (in *Build_full_history*) holds and the blue ones show how the values change according to *stride*.

To update *p_hist*, which keeps only the values that change during the iterations, we define a *Build_partial_history* function (see Figure 3.4). It takes the previous sequence, *p_hist_prev_lvl*, as an argument, and it creates a sequence that contains the values that changed according to the actual computation by summing up each pair of elements (lines 4-5). Note that, the function uses operations `head` and `tail`, where `head` returns the first element of a sequence and `tail` returns a new sequence by eliminating the first element. Figure 3.5 (middle) shows the result of applying *Build_partial_history* to *p_hist_prev_lvl*.

Down-Sweep Ghost Variables Next, in down-sweep, we define a ghost vari-

stride	*f_hist*	*p_hist*	*down_seq*
8	{1, 3, 3, 10, 5, 11, 7, 36}	{36}	{0}
4	{1, 3, 3, 10, 5, 11, 7, 26},	{10, 26},	{0, 10}
2	{1, 3, 3, 7, 5, 11, 7, 15},	{3, 7, 11, 15},	{0, 3, 10, 21}
1	{1, 2, 3, 4, 5, 6, 7, 8},	{1, 2, 3, 4, 5, 6, 7, 8},	{0, 1, 3, 6, 10, 15, 21, 28}

Figure 3.5: Ghost variables: (left) Building *f_hist* by applying *Build_full_history* to *f_hist_prev_lvl*, blue color indicates how value changes, (middle) Building *p_hist* by applying *Build_partial_history* to *p_hist_prev_lvl*, colors show combination of each pair and (right) creating *down_seq* by applying *p_sum* to *p_hist_lvl*.

able, *down_seq*, as a sequence to keep the values that change only in the current iteration. In this way, we can show that the values that change in down-sweep are in fact the exclusive prefix sum of the values changed in up sweep in each iteration. To update *down_seq* in each iteration of down-sweep, we define a function, *epsum* (Figure 3.6), and we apply it to the corresponding level of *p_hist*, shown as *p_hist_lvl* in the function. Note that the `intsum` operation sums all elements in a sequence and `take(xs, i)`, returns the i first elements of a sequence `xs`. The *epsum* function calculates the exclusive prefix sum for each element in *p_hist_lvl* and returns it as a sequence to update *down_seq*. As an example, Figure 3.5 (right) shows how *down_seq* is updated in each iteration. As we can see, the elements in *down_seq* are the exclusive prefix sum of the elements in *p_hist* at each level. Hence, it is the exclusive prefix sum of the lowest level which is the input array.

Relating Ghost Variables and Concrete Variables We proved functional correctness over the ghost variables, but we need to prove it against the actual arrays. Therefore, the last step is to relate them. First of all, it is trivial to relate the levels in *f_hist* to the output array, because of the postconditions in Figure 3.3 (lines 2-8), but we should relate the output array and *p_hist*. Figure 3.7 indicates the relationship between the output array and *p_hist*, according to *tid* and *indicator*, where gray colors (in the table) indicate the active threads in each iteration. The loop of the algorithm starts from level 1. We update the values in the output array according to the current values. Correspondingly, the values are created in *p_hist* according to the previous level. The *indicator* and *stride* are also updated in each iteration. In the output array and *p_hist*, the same colors belong to one thread according to *tid*, *indicator* and *stride*. The invariants that we

```
/*@ req 0 ≤ i && i ≤ |p_hist_lvl|;
    ens |\result| == |p_hist_lvl| - i;
    ens (\forall int j; j ≥ 0 && j < |\result|;
             \result[j] == intsum(take(p_hist_lvl, i+j))); @*/
pure seq<int> epsum_helper(seq<int> p_hist_lvl, int i) =
   i <|p_hist_lvl| ?
     seq<int> {intsum(take(p_hist_lvl, i))} + epsum_helper(p_hist_lvl, i+1) :
   seq<int> { };

/*@ ens |\result| == |p_hist_lvl|;
    ens (\forall int j; j ≥ 0 && j < |\result|;
             \result[j] == intsum(take(p_hist_lvl, j))); @*/
pure seq<int> epsum(seq<int> p_hist_lvl) = epsum_helper(p_hist_lvl, 0);
```

Figure 3.6: The *epsum* function.

Figure 3.7: Relation between Output (left) and *p_hist* (middle) according to active threads (grey color) in the table (right): $Output[indicator] = p_hist[lvl\text{-}1][2{\times}tid{+}1]$ and $Output[indicator\text{-}stride] = p_hist[lvl\text{-}1][2{\times}tid]$ ($lvl > 0$).

have in each iteration of up-sweep is $Output[indicator] = p_hist[lvl{-}1][2{\times}tid{+}1]$ and $Output[indicator{-}stride] = p_hist[lvl{-}1][2{\times}tid]$. To prove them as loop invariants in VerCors, we need some smaller steps and prove the following property:

Property 3.1 For any sequence xs:
$\forall i.\, 0 \leq i < |xs| \rightarrow Build_partial_history(xs)[i] = xs[2{\times}i] + xs[2{\times}i{+}1]$.

Proof. To prove this property in VerCors, we use *lemma functions* [VME18, GT16] whose specification captures the desired property, while the proof is encoded as a side-effect-free imperative program. Figure 3.8 shows the lemma function named *lemma_prop_1* that proves **Property 3.1**. As we can see the proof is a recursive function that calls its body in the proof (line 9). Moreover, we assert a

```
1  /*@ req |xs| % 2 == 0;
2      req 0 ≤ i && i < |Build_partial_history(xs)|;
3      req (2×i) < |xs|;
4      req (2×i+1) < |xs|;
5      ens Build_partial_history(xs)[i] == xs[2×i] + xs[2×i+1]; @*/
6  void lemma_prop_1(seq<int> xs, int i) {
7    if (1 < |xs|) {
8      if (0 < i) {
9        lemma_prop_1(tail(tail(xs)), i - 1);
10       //@ assert |Build_partial_history(xs)| == |xs| / 2;
11     }
12   }
13 }
```

Figure 3.8: The proof steps for **Property 3.1**.

property about the size of applying the *Build_partial_history* function to a sequence (line 10). This specific property is proven by induction using a different lemma function in VerCors. Note that *lemma_prop_1* proves **Property 3.1** for an arbitrary i within the domain of the universal quantified variable.

□

Using this property and the invariants, we can establish the relation between the output array and *p_hist*. Figure 3.9 shows the annotated loop of the up-sweep phase indicating some small proof steps. The assert statements are added to clarify the proof steps. By using the loop invariants (lines 1-2) and the computations of the algorithm (line 7), the relation of the output array and the previous level of *p_hist* is revealed (line 11). Then, by applying **Property 3.1** to *p_hist* (line 14) together with the fact that *p_hist* is created by applying the function *Build_partial_history* to the previous level (lines 19-21), we can relate the current level of *p_hist* and the output array (line 23).

The invariants that hold in each iteration of down-sweep is $Output[indicator] = down_seq[tid]$ and $Output[indicator - stride] = p_hist[lvl][2 \times tid]$ (see Figure 3.10, for an example). Again, the gray colors indicate the active threads and the same colors (in ghost and array) belong to one thread. To prove the invariants in the tool, we first prove these two properties:

Property 3.2 For any sequence xs:
$\forall i.\, 0 \leq i < |xs|/2 \rightarrow epsum(Build_partial_history(xs))[i] = epsum(xs)[2 \times i]$.

Proof. Figure 3.11 shows how we prove **Property 3.2** by encoding it a lemma

```
1  /*@ LI1: inv Output[indicator] == p_hist[lvl-1][2×tid+1];
2      LI2: inv Output[indicator-stride] == p_hist[lvl-1][2×tid];
3      LI3: inv (\forall int i; i > 0 && i < lvl;
4                             p_hist[i] == Build_partial_history(p_hist[i-1])); @*/
5  while(stride < N){
6    if(indicator < N && indicator ≥ stride){
7      Output[indicator] = Output[indicator] + Output[indicator − stride];
8      // By line 7:
9      //@ assert Output[indicator] == Output[indicator] + Output[indicator-stride];
10     // By LI1 and LI2:
11     //@ assert Output[indicator] == p_hist[lvl-1][2×tid+1] + p_hist[lvl-1][2×tid];
12   }
13   // By applying Property 3.1 to p_hist[lvl-1]:
14   //@ ghost lemma_prop_1(p_hist[lvl-1], tid);
15   //@ assert Build_partial_history(p_hist[lvl-1])[tid] ==
           p_hist[lvl-1][2×tid] + p_hist[lvl-1][2×tid+1];
16   // By lines 11 and 15:
17   //@ assert Output[indicator] == Build_partial_history(p_hist[lvl-1])[tid];
18
19   //@ ghost p_hist = p_hist + seq<seq<int>> Build_partial_history(p_hist[lvl-1]) ;
20   // By line 19:
21   //@ assert p_hist[lvl] == Build_partial_history(p_hist[lvl-1]);
22   // By lines 17 and 21:
23   //@ assert Output[indicator] == p_hist[lvl][tid];
24 }
```

Figure 3.9: The proof steps to relate *p_hist* levels to *Output* inside the loop of the up-sweep.

function in VerCors. The intermediate proof steps require us to prove two other properties related to using `intsum` and `take` operations in combination with the *Build_partial_history* function in VerCors (line 11 and 14). These two properties are proven by induction in VerCors encoded as separate lemma functions. Again, *lemma_prop_2* proves **Property 3.2** for an arbitrary i within the domain of the universal quantified variable. □

Property 3.3 For any sequence xs:
$\forall i.\, 0 \leq i < |xs|/2 \rightarrow epsum(xs)[2{\times}i{+}1] = epsum(xs)[2{\times}i] + xs[2{\times}i]$.

Proof. Figure 3.12 shows the proof steps for **Property 3.3**. In the intermediate proof steps, we need to prove a property of the `intsum` function (in line 14) that for any sequences xs and ys it holds that $\texttt{intsum}(xs{+}ys) == \texttt{intsum}(xs){+}\texttt{intsum}(ys)$. □

stride									
8	1	3	3	10	5	11	7	0	{ 0 }
4	1	3	3	0	5	11	7	10	{ 0, 10 }
2	1	0	3	3	5	10	7	21	{ 0, 3, 10, 21 }
1	0	1	3	6	10	15	21	28	{ 0, 1, 3, 6, 10, 15, 21, 28 }
Index:	0	1	2	3	4	5	6	7	

lvl	*tid/indicator*								*stride*
3	0/7	1/15	2/23	3/46					4
2	0/3	1/7	2/11	3/15					2
1	0/1	1/3	2/5	3/7					1
0	0/0	1/1	2/2	3/3	4/4	5/5	6/6	7/7	0

Figure 3.10: Relation between the actual array, *Output*, (left) and the ghost variable, *down_seq* (middle) according to active threads (grey color) in the table (right).

```
1  /*@ req |xs| % 2 == 0;
2      req |Build_partial_history(xs)| == |xs| / 2;
3      req 0 ≤ i && i < |Build_partial_history(xs)|;
4      req 2 × i < |xs|;
5      ens epsum(Build_partial_history(xs))[i] == epsum(xs)[2×i]; @*/
6  void lemma_prop_2(seq<int> xs, int i) {
7    // We have:
8    //@ assert epsum(Build_partial_history(xs))[i] ==
9           intsum(take(Build_partial_history(xs), i));
10   // Then we prove by induction that:
11   //@ assert intsum(take(Build_partial_history(xs), i)) ==
12          intsum(Build_partial_history(take(xs, 2×i)));
13   // Finally, we prove (by induction) that:
14   //@ assert intsum(Build_partial_history(take(xs, 2×i))) == intsum(take(xs, 2×i));
15   // By lines 8, 11 and 14 we have that:
16   //@ assert epsum(Build_partial_history(xs))[i] == intsum(take(xs, 2×i));
17 }
```

Figure 3.11: The proof steps for **Property 3.2**.

As in up-sweep, by using the invariants, the two properties and several intermediate small steps, we can establish the relation between *down_seq* and the output array. We refer to the implementation for further proof details.

3.4 Verification of Kogge-Stone's Parallel Algorithm

This section explains the verification of Kogge-Stone's parallel prefix sum algorithm. We first explain the algorithm and then we discuss how to verify it using

```
1  /*@ req 0 ≤ i;
2      req 2×i+1 < |xs|;
3      ens epsum(xs)[2×i+1] == epsum(xs)[2×i] + xs[2×i]; @*/
4  void lemma_prop_3(seq<int> xs, int i) {
5    // We have:
6    //@ assert epsum(xs)[2×i] == intsum(take(xs, 2×i));
7    //@ assert epsum(xs)[2×i+1] == intsum(take(xs, 2×i+1));
8    //@ assert take(xs, 2×i+1) == take(xs, 2×i) + seq<int> { xs[2×i] };
9
10   // Applying the intsum operation to both equation sides in line 8:
11   //@ assert intsum(take(xs, 2×i+1)) ==
12         intsum(take(xs, 2×i) + seq<int> { xs[2×i] });
13
14   // We prove by induction that intsum(xs+ys) == intsum(xs)+intsum(ys):
15   //@ assert intsum(take(xs, 2×i) + seq<int> { xs[2×i] }) ==
16         intsum(take(xs, 2×i)) + intsum(seq<int> { xs[2×i] });
17   //@ assert intsum(take(xs, 2×i) + seq<int> { xs[2×i] }) ==
18         intsum(take(xs, 2×i)) + xs[2×i];
19
20   // By lines 11 and 17 we have that:
21   //@ assert intsum(take(xs, 2×i+1)) == intsum(take(xs, 2×i)) + xs[2×i];
22   // By lines 6 and 7 we have that:
23   //@ assert epsum(xs)[2×i+1] == epsum(xs)[2×i] + xs[2×i];
24 }
```

Figure 3.12: The proof steps for **Property 3.3**.

the same approach as before. Again, we first discuss data-race freedom and then functional correctness.

3.4.1 Kogge-Stone's Prefix Sum

In contrast to Blelloch's algorithm, Kogge-Stone's [KS73] algorithm consists of one phase. Algorithm 3.2 illustrates the encoding and Figure 3.13 illustrates the algorithm visually. The levels in the figure correspond to lines 2-11 of the algorithm. In the figure, the lowest level are the input values. As we can see, at each level, each thread (*tid*) sums up elements in locations *tid* and *tid*−*offset*. Since threads need current values before updating, in the algorithm, we use an auxiliary variable, *temp*, and a barrier (line 7). The threads are synchronized at each level by another barrier (line 10). As a result, at the highest level, where *offset* exceeds the length of the array, the values are the prefix sum of the values in the input array.

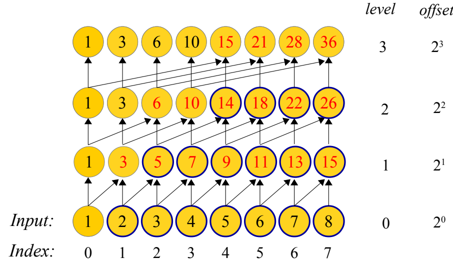

Figure 3.13: Kogge-Stone's prefix sum algorithm (two arrows coming to a circle indicates summation and one arrow indicates replacement, red color values show the effect of computations and circles with thick border show $tid \geq offset$ as in Algorithm 3.2).

Algorithm 3.2 Kogge-Stone's Prefix Sum Algorithm

```
1:  function INCLUSIVE_PREFIXSUM(int[] Input, int[] Output, int tid, int N)
2:      int offset = 1; int temp;
3:      while offset < N do
4:          temp = Output[tid];
5:          if tid ≥ offset then
6:              temp = Output[tid−offset] + temp;
7:          Barrier(tid);
8:          if tid ≥ offset then
9:              Output[tid] = temp;
10:         Barrier(tid);
11:         offset = 2×offset;
```

3.4.2 Data-Race Freedom

To verify data-race freedom of this algorithm, we need to specify permissions over the output array. Figure 3.14 shows the permission pattern in each iteration. As in Algorithm 3.2, each thread (tid) first needs read permission to locations tid and $tid-offset$ (lines 4 and 6). Since $offset$ initially is 1, each thread (tid) needs read permission to its own (tid) and its left ($tid-1$) locations as indicated by the red color in Figure 3.14. Then, in the first barrier (line 7), each thread gives up read permissions and obtains write permission to its location to store the results of the computation in line 9 (as shown in blue in Figure 3.14). Finally, threads reach the second barrier (line 10) and we change the permissions according to the new value of $offset$ for the next iteration. This is indicated in green in the figure. This pattern is repeated by each iteration of the algorithm. At the end

offset								
8	Rt_0	Rt_1	Rt_2	Rt_3	Rt_4	Rt_5	Rt_6	Rt_7
	Wt_0	Wt_1	Wt_2	Wt_3	Wt_4	Wt_5	Wt_6	Wt_7
4	Rt_0,t_4	Rt_1,t_5	Rt_2,t_6	Rt_3,t_7	Rt_4	Rt_5	Rt_6	Rt_7
	Wt_0	Wt_1	Wt_2	Wt_3	Wt_4	Wt_5	Wt_6	Wt_7
2	Rt_0,t_2	Rt_1,t_3	Rt_2,t_4	Rt_3,t_5	Rt_4,t_6	Rt_5,t_7	Rt_6	Rt_7
	Wt_0	Wt_1	Wt_2	Wt_3	Wt_4	Wt_5	Wt_6	Wt_7
1	Rt_0,t_1	Rt_1,t_2	Rt_2,t_3	Rt_3,t_4	Rt_4,t_5	Rt_5,t_6	Rt_6,t_7	Rt_7
Index:	0	1	2	3	4	5	6	7

Figure 3.14: Permissions in Kogge-Stone's algorithm; R_{t_i,t_j} indicates read permission by threads i and j, W_{t_i} indicates write permission by thread i, red color shows initial permissions, blue/green show how the permissions change in the first/second barrier.

of this algorithm, since *offset* is greater than all *tids*, each thread only has read permission to its own location (*tid*).

3.4.3 Functional Correctness

Next, we briefly discuss how to verify functional correctness of the algorithm. The difference between this algorithm and the previous one is that first, Kogge-Stone is an inclusive prefix sum algorithm and second, there is only one phase. Having one phase makes it easier to verify functional correctness, even though this algorithm is in-place as well. We could reuse the functions and operations we defined for the earlier verification. Since this algorithm is for an inclusive prefix sum, first of all, we slightly change the definition of *epsum* to be an inclusive prefix sum (*ipsum* in Figure 3.15). The strategy to verify this algorithm is the same as before, i.e., we define a ghost variable to capture the elements in the output array and a function to update this ghost variable in the same way as the actual computation does. Then, we prove functional correctness over this ghost variable by using a suitable property. Finally, we relate the ghost variable to the output array.

As we can see in Figure 3.13, in each iteration, the values from index 0 up to index *offset* are actually the inclusive prefix sum of the input array. We use this property as a loop invariant to show that at the end of the algorithm, we have the prefix sum of the input array. Thus, we define a ghost variable, *temp_seq*, and we update it inside the loop according to the *partial_prefixsum* function in Figure 3.16. This function captures the same computation as in the algorithm.

```
1  /*@ req 0 ≤ i && i ≤ |input_seq|;
2      ens |\result| == |input_seq| - i;
3      ens (\forall int j; j ≥ 0 && j <|\result|;
4                  \result[j] == intsum(take(input_seq, i+j+1))); @*/
5  pure seq<int> ipsum_helper(seq<int> input_seq, int i) =
6     i < |input_seq| ?
7        seq<int> {intsum(take(input_seq, i+1))} + ipsum_helper(input_seq, i+1) :
8     seq<int> { };
9
10 /*@ ens |\result| == |input_seq|;
11     ens (\forall int j; j ≥ 0 && j <|\result|;
12                 \result[j] == intsum(take(input_seq, j+1))); @*/
13 pure seq<int> ipsum(seq<int> input_seq) = ipsum_helper(input_seq, 0);
```

Figure 3.15: The *ipsum* function.

```
1   /*@ req |input_seq| ≥ 0 && index ≥ 0 && index ≤ |input_seq|;
2       req offset > 0 && offset ≤ 2×|input_seq|;
3       ens |\result| == |input_seq| - index;
4       ens (\forall int j; 0 ≤ j && j < |\result|; \result[j] ==
5                intsum(take(input_seq, index+j+1)) -
6                intsum(take(input_seq, index+j+1-offset))); @*/
7   static pure seq<int> partial_prefixsum(seq<int> input_seq, int index, int offset) =
8     index < |input_seq| ? seq<int> {intsum(take(input_seq, index+1)) -
9        intsum(take(input_seq, index+1-offset))} +
10       partial_prefixsum(input_seq, index+1, offset) : seq<int> { };
```

Figure 3.16: The *partial_prefixsum* function.

We can see from the postcondition of the function (lines 4-6 in Figure 3.16) that if *index* (and the corresponding *tid*) is less than *offset*, then the second `intsum` returns 0, and the first `intsum` returns the prefix sum up to *index*[3]. Thus, in each iteration for *tid* less than *offset* the result will be the prefix sum in *temp_seq*. Therefore, in the end, when *offset* is the length of the input (and output) array, all values in the ghost variable are the prefix sum of the values in the input array.

As we use *offset* in the function and from the postcondition that we defined, VerCors can infer that in each iteration for *tid* less than *offset*, *temp_seq* and the output array have the same values (specified by a loop invariant). Thus, we conclude that Kogge-Stone's algorithm indeed computes the prefix sum.

[3]Note that, the *partial_prefixsum* is a recursive function. In lines 4-6, for the final result, j is 0 and the parameter of `take` will be $index+1$, which means the first $index+1$ elements (i.e., starting from 0 it becomes up to element *index*).

```
1  /*@ req (\forall* int i; i ≥ 0 && i < gcount;
2            (\forall* int j; j ≥ 0 && j < bsize; pre(i, j)));
3      ens (\forall* int i; i ≥ 0 && i < gcount;
4            (\forall* int j; j ≥ 0 && j < bsize; post(i, j))); @*/
5  void HostCode(){
6     ⋮
7     /*@ req (\forall* int k; k ≥ 0 && k < bsize; pre(gid, k));
8         ens (\forall* int k; k ≥ 0 && k < bsize; post(gid, k)); @*/
9     par Grid(int gid = 0..gcount)
10    {
11       /*@ req pre(tid);
12           ens post(tid); @*/
13       par Block(int tid = 0..bsize)
14       {
15          ...
16       }
17    }
18 }
```

Figure 3.17: General template of CUDA thread hierarchies in PVL.

3.5 Towards CUDA Verification

The verification of the parallel Blelloch's and Kogge-Stone's algorithms as discussed above used an encoding of the algorithms in PVL. As mentioned above, PVL defines several parallel programming constructs (i.e., parallel blocks, barriers, atomics, etc) that we use to capture about GPU-style programs. In this section, we discuss how we add support to reason about CUDA programs to VerCors, by transforming CUDA programs into the parallel programming constructs of PVL. We also discuss the challenges that we encountered when verifying the CUDA versions of the two parallel prefix sum algorithms.

3.5.1 CUDA to PVL Transformation

To be able to verify CUDA programs in VerCors, the tool internally transforms a CUDA program into a PVL program. The main challenge to support verification of CUDA programs is to handle the hierarchical structure to define threads in CUDA (i.e., grids and blocks). To address this, we map a CUDA kernel into two nested PVL parallel blocks. Figure 3.17 illustrates the general template resulting from the transformation.

As we can see, the outer parallel block (i.e., *Grid*) corresponds to the number of blocks in a grid and the inner one (i.e, *Block*) indicates the number of threads

per block. The two nested parallel blocks encode the CUDA kernel that runs on a GPU and everything outside the two nested parallel blocks encodes the host code that runs on the CPU. To verify the CUDA kernel part, the user only needs to add thread-level annotations for the inner parallel block. The specifications for the outer block are inferred by VerCors as separation conjunction over all threads in a block (lines 7-8 in Figure 3.17). Similarly, the specification for the whole kernel is inferred by VerCors as a separation conjunction over all thread blocks in the grid (lines 1-4 in Figure 3.17).

Moreover, the barrier and atomic operations in CUDA are transformed into the corresponding operations in PVL. Note that the barrier always synchronizes threads inside the inner block. We refer to [BHM14, ADBH15] for more details about reasoning of GPU programs with barriers and atomic operations.

CUDA Verification Example Figure 3.18 shows how we verify the kernel part of the left rotation program in Figure 2.6 written in CUDA. We assume there is only one thread block in the grid and there are *size* threads in that block. As there are pointers in CUDA, we use $\texttt{\textbackslash pointer_index}(S, idx, \pi)$ in the specification to specify permission over a specific location *idx* that pointer S points to[4]. This CUDA example is transformed in PVL, which returns (in essence) the same annotated PVL program as in Figure 2.6.

3.5.2 CUDA Verification Challenges

Generally, a major challenge that we encounter when we verify CUDA programs is the abundance of nested quantified expressions in the specification, because the extracted pre- and postconditions of the grids and kernel are quantified over the number of threads per block and the number of blocks in a grid (lines 1-4 in Figure 3.17). These nested universal quantifiers make the reasoning about the generated proof obligations more difficult for the underlying theorem prover Z3.

To mitigate this problem, we had to extend the VerCors tool implementation with many simplification rules that during the transformation process syntacticly replace these complicated expressions by simpler ones[5]. For instance, for an array *arr* the tool applies a simplification rule that replaces

$$(\texttt{\textbackslash forall*}\ \texttt{int}\ i; i \geq 0\ \&\&\ i < gcount;\\ (\texttt{\textbackslash forall*}\ \texttt{int}\ j; j \geq 0\ \&\&\ j < bsize; \texttt{Perm}(arr, bsize \times i + j)))$$

[4] We use the predicate $\texttt{\textbackslash pointer}((S_0, \ldots, S_n), \ell, \pi)$ to indicate that all array references $S_0, \ldots, S_n$ have length ℓ, and that the current thread has permission $\pi \in (0, 1]$ for them.

[5] We would like to thank Lars van den Haak for helping to find these rules.

```
1  /*@ context_everywhere array != NULL && array.length == size;
2       req threadIdx.x != size-1 ? \pointer_index(array, threadIdx.x+1, 1\2)
3                                 : \pointer_index(array, 0, 1\2);
4       req \pointer_index(array, threadIdx.x, 1\2);
5       ens \pointer_index(array, threadIdx.x, 1);
6       ens threadIdx.x != size-1 ==> array[threadIdx.x] == \old(array[threadIdx.x+1]);
7       ens threadIdx.x == size-1 ==> array[threadIdx.x] == \old(array[0]); @*/
8  __global__ void CUDALeftRotation(int *array, int size) {
9     int tid = threadIdx.x;  // get the thread id
10    int temp;
11    if (tid != size-1) { temp = array[tid+1]; } else { temp = array[0]; }
12
13    /*@ req tid != size-1 ? \pointer_index(array, tid+1, 1\2)
14                          : \pointer_index(array, 0, 1\2);
15        req \pointer_index(array, tid, 1\2)
16        ens \pointer_index(array, tid, 1); @*/
17    __syncthreads();
18
19    array[tid] = temp;
20 }
```

Figure 3.18: Parallel left rotation in CUDA.

by

$$bsize > 0 \Rightarrow (\texttt{\textbackslash forall} * \texttt{int}\ i; i \geq 0\ \&\&\ i < gcount \times bsize; \texttt{Perm}(arr, i)).$$

The situation becomes even more complicated when there are complex access patterns to an array (e.g., $2\times tid+1$), which requires many specialized simplification rules to be added, for all different access patterns. It is future work to investigate if we can find more general simplification rules that can address a large range of array access patterns.

In particular, during the verification of the CUDA implementations of the two prefix sum algorithms in VerCors, we encountered several challenges. First, after defining the ghost variables inside the kernel, we had to specify that the initial values in those sequences are the same as the input. Unfortunately, in our current version of the VerCors verifier, it is not possible to define pure functions to initialize the sequences according to the concrete input, because pure functions only can be used for sequences and not for pointers and arrays. The second problem that we encountered is when we need to re-establish the relation between the ghost variables and the concrete ones after a barrier. We prove the relation before the barrier, but as the permission pattern changes in the barrier, we should re-establish the relation again. Unfortunately, again with the current version of the tool, we

cannot specify this in the barrier. The reason for this is that the current version of the VerCors verifier has a set-up for ghost variables that is not well-tailored to C programs (as CUDA is a variant of the C support in VerCors). There are no fundamental reasons, but adjusting this requires a major reorganization of the tool's internals. Therefore, as a temporary workaround we added a few explicit assumptions in the CUDA programs, which specify the relevant properties about the ghost variables. It should be stressed that when the PVL pseudocode version of the algorithm was verified, these properties all could be proven, thus we see no fundamental problem with adding those assumptions, it is just a practical temporary workaround.

As we use complex access patterns to an array (i.e., $2{\times}tid$ and $2{\times}tid{+}1$) in the prefix sum algorithms, the underlying theorem prover Z3 has a hard time to prove or refute complicated proof obligations with nested quantifiers, as also mentioned above. However, to be able to benefit from the synchronization algorithm on the GPU, the two prefix sum examples are implemented in such a way that the entire algorithm resides in one kernel. The consequence of this design choice is that we can only have one thread block, which is restricted to a limited number of threads. That means, the input size is restricted to that limit. However, the advantage of this for verification is that instead of using `threadIdx.x`+`blockIdx.x`×`blockDim.x`, we can use `threadIdx.x` as thread identifiers. This simplifies proof obligations and mitigates the problem in Z3. To be able to do this, we explicitly specify that the number of thread blocks is one in the contract of the kernel. In this way, we further simplify the generated proof obligations in Z3.

In general, we find that specifying the number of thread blocks and threads per block explicitly in the contract simplifies the complexity, as we can replace thread block variable (e.g., *gcount*) and size of blocks (*bsize*) by concrete values in the proof. Note that these are necessarily concrete user-specified parameters when invoking a CUDA kernel.

3.6 Related Work

The closest related work to our work in this chapter is by Chong et al. [CDK14] where they verify data-race freedom and propose a method to verify functional correctness of Blelloch's and Kogge-Stone's algorithm along with two other parallel prefix sum algorithms for all inputs *up to fixed sizes*. They show that if a parallel prefix sum algorithm is proven to be data race-free, then the correctness can be established by generating one test case. Therefore, they use GPUVerify to prove data-race freedom of 4 parallel prefix sum algorithms. Their approach is applicable for any parallel prefix sum algorithm with other operations and types

instead of summation and integers. Comparing VerCors to their tool, GPUVerify benefits from more automation, while we need to specify the annotations manually. However, using GPUVerify to verify data-race freedom of GPU programs, the input size must be bounded. As a result, they only show functional correctness for *a fixed input size* (a realistic size for current GPUs). In this chapter, we verified data-race freedom and also functional correctness of the two algorithms for *any arbitrary size of input*. We believe that it should be no problem to also prove the other two algorithms.

3.7 Conclusion

This chapter shows how we verify data-race freedom and functional correctness of the two most widely-used parallel prefix sum algorithms, Blelloch's and Kogge-Stone's algorithm, for *an arbitrary input size* by encoding the algorithms into VerCors verifier. Proving functional correctness of Blelloch's algorithm is challenging for multiple reasons. First, the algorithm is in-place. Second, it consists of two independent, but related phases and third, it is non-trivial to relate the computations in both phases to conclude the desired end result (i.e., that it establishes a prefix sum). We overcome these challenges by introducing ghost variables and defining suitable functions that mimic the computations on the ghost variables. Moreover, we prove suitable properties that help us to reason about the algorithm. The verification of Kogge-Stone's algorithm is not as hard as the first one, since there is only one phase and the property that we define is straightforward. We benefit from functions, operations and properties that are defined in the earlier verification and reuse them in the second verification. Finally, we show how to add CUDA support in VerCors and we discuss challenges in the verification of CUDA versions of the two algorithms.

As future work, we can verify more complicated parallel algorithms that use the prefix sum algorithm internally, such as sorting algorithms. We also would like to investigate how to further automate the process of proof creation. We believe that a substantial part of the required annotations, in particular those related to permissions, can be generated automatically.

CHAPTER 4

Verification of Parallel Stream Compaction and Summed-Area Table Algorithms

This chapter studies two applications of prefix sum algorithms described in the previous chapter. Concretely, Stream Compaction (SC) algorithm uses Blelloch's algorithm, and Summed-Area Table (SAT) algorithm uses Kogge-Ston's prefix sum. Therefore, this chapter takes advantage of the correctness proof for the prefix sums to prove the correctness (i.e., data-race freedom and functional correctness) of SC and SAT algorithms. It demonstrates how we can reuse a verified sub-function (i.e., prefix sum) to prove more complicated algorithms (i.e., stream compaction and summed area table) in a modular way with less effort. Moreover, it shows that it is feasible in practice to verify larger case studies by building the verification of the complicated algorithm on top of the basic one.

4.1 Introduction

Stream compaction and summed-area table [Cro84] algorithms are another examples where the parallel (GPU-based) implementations [Hor05, HSO07, HSC+05, BOA09, KNI14] outperform the sequential (CPU-based) counterparts. Stream compaction reduces an input array to a smaller array by removing undesired elements. This is an important operation on GPUs, because a variety of applications such as collision detection and sparse matrix compression rely on it. The reduction in size by eliminating undesired elements is useful because (1) the computation can be done more efficiently by not wasting the computation power on undesired elements and, (2) it greatly reduces the transfer costs between the CPU and GPU, especially for applications where data transfer between CPU and GPU is frequent.

A summed-area table is a two-dimensional (2D) table generated from a 2D input array where each entry in the table is the sum of all values in the square defined by

the entry location and the upper-left corner in the input array. Generating such a table is useful in computer graphics and image processing [HSC+05].

The two algorithms use a prefix sum algorithm, which takes an array of integers and, for each element, computes the sum of the previous elements. In the previous chapter, we verified data-race freedom and functional correctness of two parallel prefix sum algorithms (i.e., Blelloch's and Kogge-Stone's algorithms). In this chapter, we investigate how we can benefit from the already verified prefix sum algorithms to prove the stream compaction and summed-area table algorithms; i.e., how much effort is needed to adapt the specifications from the verified prefix sums for the verification of these two algorithms.

Even though we could build the verification on top of the earlier proofs, proving functional correctness of SC and SAT is still challenging because (1) in the stream compaction algorithm, the input of the prefix sum sub-function is an array of flags and the output is used as indices of elements in another array. Therefore, additional properties should be proved to reason about the prefix sum result to be safely used as indices; and (2) in the summed-area table algorithm, in addition to the prefix sum, the transposition operation is used intermittently. Due to these intermediate steps, we should store the manipulated values and establish a formal relation between the values of each step in order to reason about the final result (i.e, output).

Our approach is to use deductive verification to prove correctness of parallel SC and SAT algorithms using VerCors. We show that the verification of larger case studies is feasible, by adding the verification of the more complex algorithm on top of the basic one, not only in theory, but also in practice using tool support. Moreover, our work enables the verification of other parallel algorithms that are built on top of the stream compaction and summed-area table algorithms, such as collision detection [TMLT11, GGK06] and box filtering [Cro84, KHB09].

In the rest of this chapter, we explain the verification of stream compaction and summed-area table algorithms. Concretely, we explain each algorithm and its enccoding in the VerCors verifier. Then, we prove data-race freedom and functional correctness of each algorithm.

4.2 Verification of Parallel Stream Compaction Algorithm

This section describes the stream compaction algorithm and how we verify it. First, we explain the algorithm and its encoding in VerCors. Then, we prove data-race freedom and we show how we prove functional correctness of the algorithm. Moreover, we show how we can reuse the verified Blelloch's parallel prefix

Algorithm 4.1 Stream Compaction Algorithm

```
1: function STREAM_COMPACTION(int[] Input, int[] Output, int[] Flag, int[] ExPre, int
   N)
2:   Par(tid = 0..N)
3:     EXCLUSIVE_PREFIXSUM(Flag, ExPre, tid, N);
4:     Barrier(tid);
5:     if Flag[tid] == 1 then
6:         Output[ExPre[tid]] = Input[tid];
```

sum algorithm from the previous chapter to reason about the stream compaction algorithm.

4.2.1 Stream Compaction Algorithm

Given an array of integers as input and an array of booleans that flag which elements are desired, stream compaction returns an array that holds only those elements of the input whose flags are true. An algorithm is a stream compaction if it satisfies the following:

- INPUT: two arrays, *Input* of integers and *Flag* of booleans of size N where M $(M \leq N)$ elements are true.
- OUTPUT: an array *Output* such that
  - $|Output| = M$.
  - $\forall j.\ 0 \leq j < M\colon Output[j] = t \Rightarrow \exists i.\ 0 \leq i < N\colon Input[i] = t \ \wedge \ Flag[i]$.
  - $\forall i.\ 0 \leq i < N\colon Input[i] = t \ \wedge \ Flag[i] \Rightarrow \exists j.\ 0 \leq j < M\colon Output[j] = t$.

Algorithm 4.1 shows the pseudocode of the parallel algorithm and Figure 4.1 presents an example of stream compaction. Initially we have an input and a flag array (implemented as integers of zeros and ones). To keep the flagged elements and discard the rest, first we calculate the exclusive prefix sum (as in the previous chapter) of the flag array. Interestingly, for the elements whose flags are 1, the exclusive prefix sum indicates their location (index) in the output array. In the implementation, the input of the prefix sum function is *Flag* and the output is stored in *ExPre* (line 3). Then all threads are synchronized by the barrier in line 4, after which all the desired elements are stored in the output array (lines 5-6).

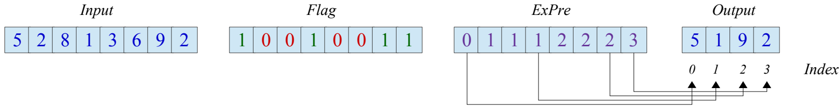

Figure 4.1: An example of stream compaction of size 8.

Arrays \ Locations	At the beginning of the algorithm	After the exclusive prefix sum	After the barrier
Input	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7
Flag	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7
ExPre	Wt_0 Wt_0 Wt_1 Wt_1 Wt_2 Wt_2 Wt_3 Wt_3	Wt_0 Wt_1 Wt_2 Wt_3 Wt_4 Wt_5 Wt_6 Wt_7	Rt_0 Rt_1 Rt_2 Rt_3 Rt_4 Rt_5 Rt_6 Rt_7
Output	Wt_0 Wt_1 Wt_2 Wt_3	Wt_0 Wt_1 Wt_2 Wt_3	Wt_f Wt_f Wt_f Wt_f
Index	0 1 2 3	0 1 2 3	0 1 2 3

Figure 4.2: Permission pattern of arrays in stream compaction algorithm corresponding to Figure 4.1; Rt_i / Wt_i means thread i has read/write permission. Green color indicates permission changes.

4.2.2 Data-Race Freedom

To prove data-race freedom, we specify how threads access shared resources by adding permission annotations to the code. In Algorithm 4.1, we have several arrays that are shared among threads. There are three locations in the algorithm where permissions can be redistributed: before Algorithm 4.1 as preconditions, in the exclusive prefix sum function as postconditions and in the barrier (redistribution of permissions). Figure 4.2 visualizes the permission pattern for those shared arrays, which reflects the permission annotations in the code according to these three locations. The explanation of the permission patterns in each array in these three locations is as follows:

- *Input*: since each thread (*tid*) only needs read permission (line 6 in Algorithm 4.1), we define each thread to have read permissions to its "own" location at index *tid* throughout the algorithm (Figure 4.2). This also ensures that the values in *Input* cannot be changed.
- *Flag*: since *Flag* is the input of the exclusive prefix sum function, its permission pattern at the beginning of Algorithm 4.1 must match the permission preconditions of the exclusive prefix sum function. Thus, following the preconditions of this function (see the previous chapter), we define the permissions such that each thread (*tid*) has read permissions to its "own" location (Figure 4.2: left). The exclusive prefix sum function returns the same per-

missions for *Flag* in its postconditions (Figure 4.2: middle). Since, each thread needs read permission in line 5 of Algorithm 4.1, we keep the same permission pattern in the barrier as well (Figure 4.2: right).

- *ExPre*: since *ExPre* is the output of the exclusive prefix sum function, the permission pattern at the beginning of Algorithm 4.1 should match the permission preconditions of the exclusive prefix sum function (specified in the previous chapter). Thus, each thread (tid < half *ExPre* size) has write permissions to locations $2\times tid$ and $2\times tid+1$ (Figure 4.2: left). As postcondition of the exclusive prefix sum function (specified in the previous chapter), each thread has write permission to its "own" location in *ExPre* (Figure 4.2: middle). Since each thread only needs read permission in line 6 of Algorithm 4.1, we change the permission pattern from write to read in the barrier (Figure 4.2: right).
- *Output*: it is only used in line 6 of Algorithm 4.1 and its permissions are according to the values in *ExPre*. Thus, the initial permissions for *Output* can be arbitrary and in the barrier, we specify the permissions such that each thread (tid) has write permission in location $ExPre[tid]$ if its flag is 1 (indicated by t_f in Figure 4.2: right).

4.2.3 Functional Correctness

Proving functional correctness of the parallel stream compaction algorithm consists of two parts. First, we prove that the elements in the exclusive prefix sum function (*ExPre*) are in the range of the output, thus they can be used safely as indices in *Output* (i.e., line 6 in Algorithm 4.1). Second, we prove that if the flag of an element (in *Input*) is 1, then that element is in *Output*. Moreover, we prove that the length of *Output* is the sum of elements in the flag. This implies that *Output* does not contain an element from a location in *Input* whose flag is not 1[1].

To prove both parts, we use ghost variables. Concretely, we define two ghost variables, *inp_seq* and *flag_seq* as sequences of integers to capture all values in arrays *Input* and *Flag*, respectively. Since values in *Input* and *Flag* do not change during the algorithm[2], *inp_seq* and *flag_seq* are always the same as *Input* and *Flag*[3].

First, to reuse of the exclusive prefix sum specification (line 3 in Algorithm 4.1) from the previous chapter, we should consider two points: (1) the input to the exclusive prefix sum (*Flag*) in this paper is restricted to 0 and 1; and (2) the elements in the exclusive prefix sum function (*ExPre*) should be safely usable as

[1] Note that there might be duplicated elements in *Input* whose flags are not the same.
[2] Note that threads only have read permissions over *Input* and *Flag*.
[3] Thus, properties for *inp_seq* and *flag_seq* also hold for *Input* and *Flag*.

```
1  /*@ req (\forall int j; 0 ≤ j && j < |flag_seq|; flag_seq[j]==0 || flag_seq[j]==1);
2      ens intsum(flag_seq) ≥ 0; @*/
3  void lemma_prop_1(seq<int> flag_seq){
4    if (0 < |flag_seq|) {
5      //@ assert head(flag_seq) ≥ 0;
6      lemma_prop_1(tail(flag_seq));
7    }
8  }
```

Figure 4.3: The proof steps for **Property 4.1**.

```
1  /*@ req i ≤ |flag_seq|;
2      req (\forall int j; 0 ≤ j && j < |flag_seq|; flag_seq[j]==0 || flag_seq[j]==1);
3      ens 0 ≤ intsum(take(flag_seq, i)) &&
4          intsum(take(flag_seq, i)) ≤ intsum(flag_seq); @*/
5  void lemma_prop_2(seq<int> flag_seq, int i){
6    if (0 < i) {
7      lemma_prop_2(tail(flag_seq), i - 1);
8      // By the property that intsum(xs+ys) == intsum(xs)+intsum(ys):
9      //@ assert intsum(seq<int> { head(flag_seq) } + take(tail(flag_seq), i - 1)) ==
           intsum(seq<int> { head(flag_seq) }) + intsum(take(tail(flag_seq), i - 1));
10     //@ assert intsum(seq<int> { head(flag_seq) }) == head(flag_seq);
11   } else {
12     //@ assert 0 ≤ intsum(flag_seq);
13   }
14 }
```

Figure 4.4: The proof steps for **Property 4.2**.

indices in *Output* (i.e., line 6 in Algorithm 4.1). Therefore, we use VerCors to prove some suitable properties to reason about the values of the prefix sum of the flag. The first property that we prove in VerCors is that the sum of a sequence of zeros and ones is non-negative[4]:

Property 4.1 $(\forall i.\, 0 \leq i < |flag_seq|\colon\ flag_seq[i] = 0\ \vee\ flag_seq[i] = 1) \Rightarrow intsum(flag_seq) \geq 0.$

Proof. Figure 4.3 shows the lemma function to prove **Property 4.1** in VerCors. By induction over the elements in the sequence, we assert (in line 5) that the first element is greater than or equal to zero. □

[4]The `intsum` operation sums up all elements in a sequence.

We need **Property 4.1** since the prefix sum for each element is the sum of all previous elements. We benefit from the first property to prove in VerCors that all the elements in the exclusive prefix sum of a sequence *flag_seq* (only zeros and ones) are greater than or equal to zero and less than or equal to the sum of elements in *flag_seq*[5]:

Property 4.2 $(\forall i.\, 0 \leq i < |flag_seq|\colon\ flag_seq[i] = 0\ \vee\ flag_seq[i] = 1) \Rightarrow$
$(\forall i.\, 0 \leq i < |epsum(flag_seq)|\colon\ epsum(flag_seq)[i] \geq 0\ \wedge$
$epsum(flag_seq)[i] \leq intsum(flag_seq)).$

Proof. Figure 4.4 shows the encoded lemma to prove **Property 4.2**. In the proof steps, we benefit from the `intsum` property (in line 8) that we already proved in the previous chapter. Concretely, it states that for any sequences *xs* and *ys* it holds that `intsum`$(xs{+}ys)$ == `intsum`(xs)+`intsum`(ys). Note that *lemma_prop_2* proves **Property 4.2** for an arbitrary i within the domain of the universal quantified variable. □

This gives the lower and upper bound of elements in the prefix sum, which are used as indices in *Output*. This property is not sufficient to prove that the elements are in the range of *Output* due to two reasons. First, an element in the prefix sum can be as large as the sum of ones in the flag. Hence, it might exceed *Output* size which is in the range 0 to *intsum*(*flag_seq*)-1. Second, we only use the elements in the prefix sum whose flags are 1. **Property 4.2** does not specify those elements explicitly. Therefore, we prove another property in VerCors to explicitly specify the elements in the prefix sum whose flags are 1 as follows:

Property 4.3 $(\forall i.\, 0 \leq i < |flag_seq|\colon\ flag_seq[i] = 0\ \vee\ flag_seq[i] = 1) \Rightarrow$
$(\forall i.\, 0 \leq i < |epsum(flag_seq)|\ \wedge\ flag_seq[i] = 1\colon$
$(epsum(flag_seq)[i] \geq 0\ \wedge\ epsum(flag_seq)[i] < intsum(flag_seq))).$

Proof. Figure 4.5 shows the lemma to prove **Property 4.3** in VerCors. The proof steps are similar to Figure 4.4. □

Property 4.3 guarantees that the elements in the prefix sum whose flags are 1 are truly in the range of *Output*, and can be used safely as indices. Moreover, it has been proven in the previous chapter that *epsum*(*flag_seq*) is equal to the result of the prefix sum function (i.e., *ExPre*).

Second, we reason about the final values in *Output*, similar to what we did in the previous chapter using the following steps:

1. Define a ghost variable as a sequence.

[5] The `epsum` operation of a sequence returns an exclusive prefix sum of that sequence.

```
1  /*@ req 0 ≤ i && i < |flag_seq|;
2      req (\forall int j; 0 ≤ j && j < |flag_seq|; flag_seq[j]==0 || flag_seq[j]==1);
3      req flag_seq[i] == 1;
4      ens 0 ≤ intsum(take(flag_seq, i)) &&
5          intsum(take(flag_seq, i)) < intsum(flag_seq); @*/
6  void lemma_prop_3(seq<int> flag_seq, int i){
7    if (0 < i) {
8      lemma_prop_3(tail(flag_seq), i - 1);
9      // By the property that intsum(xs+ys) == intsum(xs)+intsum(ys):
10     //@ assert intsum(seq<int> { head(flag_seq) } + take(tail(flag_seq), i - 1)) ==
           intsum(seq<int> { head(flag_seq) }) + intsum(take(tail(flag_seq), i - 1));
11     //@ assert intsum(seq<int> { head(flag_seq) }) == head(flag_seq);
12   } else {
13     // By the property that intsum(xs+ys) == intsum(xs)+intsum(ys):
14     //@ assert intsum(seq<int> { head(flag_seq) } + tail(flag_seq)) ==
           intsum(head(flag_seq)) + intsum(tail(flag_seq));
15     //@ assert 0 ≤ intsum(flag_seq);
16   }
17 }
```

Figure 4.5: The proof steps for **Property 4.3**.

2. Define a mathematical function that updates the ghost variable according to the actual computation of the algorithm.
3. Prove functional correctness over the ghost variables by defining a suitable property.
4. Relate the ghost variable to the concrete variable; i.e., prove that the elements in the ghost sequence are the same as in the actual array.

Following this approach, we define a ghost variable, *out_seq*, as a sequence of integers and a mathematical function, *filter*, as shown in Figure 4.6. This function computes the compacted list of an input sequence, *inp_seq*, by filtering it according to a flag sequence, *flag_seq* (where `head` returns the first element of a sequence and `tail` returns a new sequence by eliminating the first element). Thus, for each element in *inp_seq*, this function checks its flag to either add it to the result (line 6) or discard it (line 7). The function specification has two preconditions: (1) the length of both sequences is the same (line 1) and (2) each element in *flag_seq* is either 0 or 1 (lines 2). The postcondition states that the length of the compacted list (result) is the sum of all elements in *flag_seq* (line 3) which is at most the same length as *flag_seq* (line 4). We apply the *filter* function to *inp_seq* and *flag_seq* (as ghost statements) at the end of Algorithm 4.1 to update *out_seq*.

To reason about the values in *out_seq* and relate it to *inp_seq* and *flag_seq* we prove the following property in VerCors:

```
1  /*@ req |inp_seq| == |flag_seq|;
2      req (\forall int i; 0 ≤ i && i < |flag_seq|;
3                                          flag_seq[i] == 0 || flag_seq[i] == 1);
4      ens |\result| == intsum(flag_seq);
5      ens 0 ≤ |\result| && |\result| ≤ |flag_seq|; @*/
6  pure seq<int> filter(seq<int> inp_seq, seq<int> flag_seq) = |inp_seq|>0 ?
7    head(flag_seq)==1 ? seq<int>{head(inp_seq)} + filter(tail(inp_seq), tail(flag_seq))
8    : filter(tail(inp_seq), tail(flag_seq)) : seq<int>{};
```

Figure 4.6: The *filter* function.

Property 4.4 $(\forall i.\, 0 \leq i < |flag_seq|\colon\ flag_seq[i] = 0 \ \vee\ flag_seq[i] = 1) \Rightarrow$
$(\forall i.\, 0 \leq i < |epsum(flag_seq)| \ \wedge\ flag_seq[i] = 1\colon$
$(inp_seq[i] = filter(inp_seq, flag_seq)[epsum(flag_seq)[i]]))$.

Proof. Figure 4.7 shows the encoded lemma in VerCors to prove **Property 4.4**. As we can see, we benefit from **Property 4.2** and **Property 4.3** in the proof steps (lines 9 and 15, respectively). We also benefit from the `intsum` property in line 19. □

From **Property 4.4**, we can prove in VerCors that all elements in inp_seq (and *Input*) whose flags are 1 are in out_seq. Since we specify that the length of out_seq is the sum of all elements in the flag, which is the number of ones (line 4 in Figure 4.6), we also prove that *Output* does not contain an element from a location in *Input* whose flag is not 1.

The last step is to relate out_seq to *Output*. Figure 4.8 shows the proof steps which are located at the end of Algorithm 4.1. Through some smaller steps, and using **Property 4.4** we prove in VerCors that out_seq and *Output* is the same (line 10). Note that for each *tid*, $epsum(flag_seq)[tid]$ is equal to $ExPre[tid]$ as proven in the previous chapter.

As we can see in this verification, we could reuse the specification of the verified prefix sum algorithm, by proving some more properties. We should note that the time we spent to verify the stream compaction algorithm is much less than the verification of the prefix sum algorithm.

```
1  /*@ req 0 ≤ i && i < |flag_seq|;
2      req |flag_seq| == |inp_seq|;
3      req (\forall int j; 0 ≤ j && j < |flag_seq|; flag_seq[j]==0 || flag_seq[j]==1);
4      req flag_seq[i] == 1;
5      ens inp_seq[i] == filter(inp_seq, flag_seq)[epsum(flag_seq)[i]] @*/
6  void lemma_prop_4(seq<int> inp_seq, seq<int> flag_seq, int i){
7    if (0 < i) {
8      lemma_prop_4(tail(inp_seq), tail(flag_seq), i - 1);
9      // By applying Property 4.2:
10     //@ ghost lemma_prop_2(flag_seq, i);
11     //@ assert 0 ≤ intsum(take(flag_seq, i)) &&
           intsum(take(flag_seq, i)) ≤ intsum(flag_seq);
12
13     //@ assert intsum(take(flag_seq, i)) < |flag_seq|;
14
15     // By applying Property 4.3:
16     //@ ghost lemma_prop_3(flag_seq, i);
17     //@ assert 0 ≤ intsum(take(flag_seq, i)) &&
           intsum(take(flag_seq, i)) < intsum(flag_seq);
18
19     // By the property that intsum(xs+ys) == intsum(xs)+intsum(ys):
20     //@ assert intsum(seq<int> { head(flag_seq) } + take(tail(flag_seq), i - 1)) ==
           intsum(seq<int> { head(flag_seq) }) + intsum(take(tail(flag_seq), i - 1));
21     //@ assert intsum(seq<int> { head(flag_seq) }) == head(flag_seq);
22   } else {
23     //@ assert inp_seq[0] == filter(inp_seq, flag_seq)[intsum(take(flag_seq, 0))];
24   }
25 }
```

Figure 4.7: The proof steps for **Property 4.4**.

```
1  //@ ghost seq<int> out_seq = filter(inp_seq, flag_seq);
2  //@ assert |out_seq| == intsum(flag_seq); // By line 4 in Figure 4.6
3  if(flag_seq[tid] == 1){
4     // By applying Property 4.4:
5     //@ ghost lemma_prop_4(inp_seq, flag_seq, tid);
6     //@ assert inp_seq[tid] == filter(inp_seq, flag_seq)[epsum(flag_seq)[tid]];
7     //@ assert out_seq == filter(inp_seq, flag_seq); // By line 1
8     //@ assert inp_seq[tid] == out_seq[epsum(flag_seq)[tid]]; // By lines 5-6
9     //@ assert Output[ExPre[tid]] == Input[tid]; // By lines 6-7 in Algorithm 4.1
10    //@ assert Output[ExPre[tid]] == out_seq[epsum(flag_seq)[tid]]; // By lines 8-9
11 }
```

Figure 4.8: The proof steps to relate *out_seq* to *Output* array.

Algorithm 4.2 Summed-Area Table Algorithm

```
1:  function SUMMED_AREA_TABLE(int[][] Input, int[][] Temp, int[][] Output, int N)
2:    for(int i = 0; i < N; i++)
3:      Par(tid = 0..N)                                         // First Prefix Sum
4:        INCLUSIVE_PREFIXSUM(Input[i], Temp[i], inp_seq[i], tmp1_seq[i], tid, N);
5:    Properties 4.5 and 4.6 (Table 4.1) hold here
6:    Par(tidI = 0..N, tidJ = 0..N)                             // First Transposition
7:      int temporary = Temp[tidI][tidJ];
8:      Barrier(tidI, tidJ);
9:      Temp[tidJ][tidI] = temporary;
10:   tmp2_seq = transpose(tmp1_seq, 0, N);
11:   Properties 4.7 and 4.8 (Table 4.1) hold here
12:   for(int i = 0; i < N; i++)
13:     Par(tid = 0..N)                                         // Second Prefix Sum
14:       INCLUSIVE_PREFIXSUM(Temp[i], Output[i], tmp1_seq[i], tmp3_seq[i], tid, N);
15:   Properties 4.9 and 4.10 (Table 4.1) hold here
16:   Par(tidI = 0..N, tidJ = 0..N)                             // Second Transposition
17:     int temporary = Output[tidI][tidJ];
18:     Barrier(tidI, tidJ);
19:     Output[tidJ][tidI] = temporary;
20:   out_seq = transpose(tmp3_seq, 0, N);
21:   Properties 4.11 and 4.12 (Table 4.1) hold here
```

4.3 Verification of Parallel Summed-Area Table Algorithm

This section discusses the summed-area table algorithm and its verification. As above, after describing the algorithm and its encoding in VerCors, we first prove data-race freedom and then explain how we prove functional correctness. We also show how we reuse the verified Kogge-Stone's prefix sum from the previous chapter in the verification of the summed-area table algorithm.

4.3.1 Summed-Area Table Algorithm

Given a 2D array of integers, the summed-area table is a 2D array with the same size where each entry in the output is the sum of all values in the square defined by the entry location and the upper-left corner in the input. The algorithm's input and output are specified in the following way:

- INPUT: a 2D array *Input* of integers of size $N \times M$.

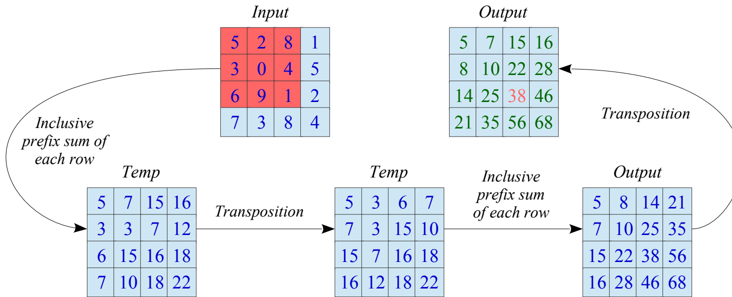

Figure 4.9: An example of summed-area table of size 4×4.

- OUTPUT: a 2D array *Output* of size $N \times M$ such that

$$Output[i][j] = \sum_{t=0}^{i} \sum_{k=0}^{j} Input[t][k] \text{ for } 0 \leq i < N \text{ and } 0 \leq j < M.$$

Algorithm 4.2 shows the (annotated) pseudocode of the parallel algorithm and Figure 4.9 shows an example for the summed-area table algorithm[6]. For example, the red element 38 in *Output* is the sum of the elements in the red square in *Input*. We apply the inclusive prefix sum (from the previous chapter) to each row of *Input* and store it in *Temp* (lines 2-4). Then, we transpose the *Temp* matrix (lines 6-9). Thereafter, we apply again the inclusive prefix sum to each row of *Temp* (lines 12-14). Finally, we transpose the matrix again, resulting in matrix *Output* (lines 16-19). The parallel transpositions after each prefix sum are determined by creating 2D thread blocks for each element of the matrix (lines 6-9 and 16-19) where each thread $(tidI, tidJ)$ stores its value into location $(tidJ, tidI)$ by first writing into a *temporary* variable (lines 7 and 17) and then synchronizing in the barrier (lines 8 and 18).

4.3.2 Data-Race Freedom

Since data-race freedom of the parallel inclusive prefix sum has been verified in the previous chapter, we only show data-race freedom of the transposition phases in Algorithm 4.2. As an example, Figure 4.10 illustrates the permission pattern in a matrix *Temp* (and also *Output*) of size 4×4. Before the barrier (lines 7 and 14 in Algorithm 4.2) each thread $(tidI, tidJ)$ has read permission in location $(tidI, tidJ)$ in *Temp* (and also *Output*). In the barrier, the permission pattern changes such that each thread $(tidI, tidJ)$ has write permission to location $(tidJ, tidI)$. In this way, each thread $(tidI, tidJ)$ can read its value from location $(tidI, tidJ)$ (before

[6] For simplicity, we consider square matrices in this thesis.

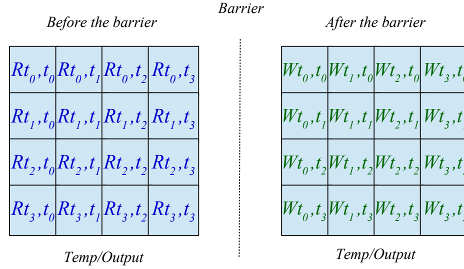

Figure 4.10: Permission pattern of matrix Temp/Output before and after the barrier in the transposition phase; $Rt_i, t_j\,/\,Wt_i, t_j$ means thread $(i,\, j)$ has read/write permission.

the barrier) and write that value into location $(tidJ,\, tidI)$ (after the barrier) safely.

4.3.3 Functional Correctness

Next, we discuss functional correctness of the parallel summed-area table algorithm. The approach to verify this algorithm is the same as before. First of all, we define two ghost variables: *inp_seq* is a sequence of sequences that captures the elements in *Input*, and *tmp1_seq* stores the inclusive prefix sum of elements in *Input* (see Figure 4.11: step 1).

After applying the first prefix sum function, **Properties 4.5** and **4.6** from Table 4.1 hold (line 5 in Algorithm 4.2). **Property 4.5** specifies that *tmp1_seq* contains the inclusive prefix sum of the elements in *inp_seq*[7]. **Property 4.6** shows the relation between *tmp1_seq*, and the actual array, *Temp*. We obtain these properties from the postconditions of the verified inclusive prefix sum (see the previous chapter).

Now, we define a mathematical function *transpose*, as shown in Figure 4.12. This function computes the transposition of a sequence of sequences. The *transpose* function creates a sequence for each column i (starting from 0) as a new row i in the result, using a helper function (*transpose_helper*) to collect the *ith* elements of each row (in the input sequence)[8]. The preconditions of both functions specify the length of the input sequence and the range of the parameter's integer variables (lines 1-2 and 10-11). The postcondition of the *transpose_helper* function (lines 12-13) indicates that the result has the same size as each row and the return sequence contains the *ith* element of each row in the input sequence. The postcondition of

[7]The `ipsum` operation of a sequence returns an inclusive prefix sum of that sequence.
[8]Both functions are recursive and they are invoked with i and k equal to zero.

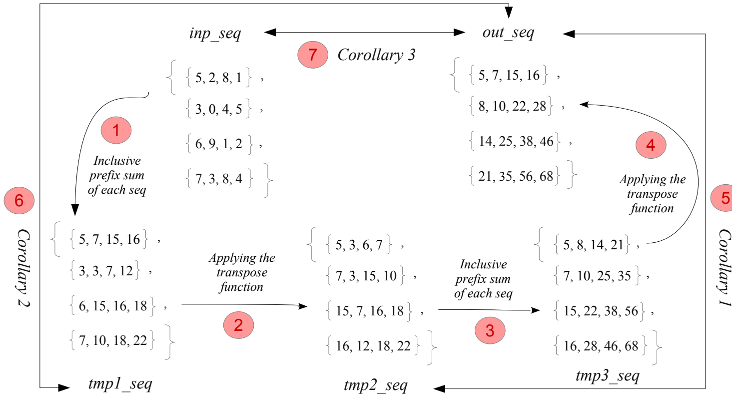

Figure 4.11: An example of phases in the summed-area table algorithm over sequences as ghost variables. The sequences and functions capture the same arrays and computations as in Figure 4.9.

```
1  /*@ req |xs| == N && (\forall int j; 0 ≤ j && j < N; |xs[j]| == N);
2      req i ≥ 0 && i ≤ N;
3      ens |\result| == N - i;
4      ens (\forall int j; 0 ≤ j && j < N - i; |\result[j]| == N);
5      ens (\forall int j; 0 ≤ j && j < N - i;
6               (\forall int k; 0 ≤ k && k < N; \result[j][k] == xs[k][i+j])); @*/
7  pure seq<int> transpose(seq<seq<int>> xs, int i, int N) = i < N ?
8      seq<seq<int>> {transpose_helper(xs, 0, i, N)} + transpose(xs, i+1, N) :
9      seq<int> {};
10
11 /*@ req |xs| == |N| && (\forall int j; 0 ≤ j && j < N; |xs[j]| == N);
12     req k ≥ 0 && k ≤ N && i ≥ 0 && i < N;
13     ens |\result| == N - k;
14     ens (\forall int j; 0 ≤ j && j < |\result|; \result[j] == xs[k+j][i]); @*/
15 pure seq<int> transpose_helper(seq<seq<int>> xs, int k, int i, int N) =
16     k < N ? seq<int> {xs[k][i]} + transpose_helper(xs, k+1, i, N) : seq<int> {};
```

Figure 4.12: The *transpose* function.

the *transpose* function (lines 3-6) indicates that the result has the same size and indeed is the transposition of the input sequence.

We apply the *transpose* function to *tmp1_seq* right after the first transposition computation of the algorithm (line 10 in Algorithm 4.2). We store the result in a different sequence as *tmp2_seq*. Figure 4.11: step 2, illustrates this phase of the

No.	Mathematical description of properties
4.5	$(\forall i. 0 \leq i < \lvert inp_seq \rvert$: $(\forall j. 0 \leq j < \lvert inp_seq[i] \rvert$: $tmp1_seq[i][j] = ipsum(inp_seq[i])[j] = \sum_{t=0}^{j} inp_seq[i][t]))$
4.6	$(\forall i. 0 \leq i < \lvert tmp1_seq \rvert$: $(\forall j. 0 \leq j < \lvert tmp1_seq[i] \rvert$: $tmp1_seq[i][j] = Temp[i][j]))$
4.7	$(\forall i. 0 \leq i < \lvert tmp1_seq \rvert$: $(\forall j. 0 \leq j < \lvert tmp1_seq[i] \rvert$: $tmp2_seq[i][j] = tmp1_seq[j][i]))$
4.8	$(\forall i. 0 \leq i < \lvert tmp2_seq \rvert$: $(\forall j. 0 \leq j < \lvert tmp2_seq[i] \rvert$: $tmp2_seq[i][j] = Temp[i][j]))$
4.9	$(\forall i. 0 \leq i < \lvert tmp2_seq \rvert$: $(\forall j. 0 \leq j < \lvert tmp2_seq[i] \rvert$: $tmp3_seq[i][j] = ipsum(tmp2_seq[i])[j] = \sum_{t=0}^{j} tmp2_seq[i][t]))$
4.10	$(\forall i. 0 \leq i < \lvert tmp3_seq \rvert$: $(\forall j. 0 \leq j < \lvert tmp3_seq[i] \rvert$: $tmp3_seq[i][j] = Output[i][j]))$
4.11	$(\forall i. 0 \leq i < \lvert tmp3_seq \rvert$: $(\forall j. 0 \leq j < \lvert tmp3_seq[i] \rvert$: $out_seq[i][j] = tmp3_seq[j][i]))$
4.12	$(\forall i. 0 \leq i < \lvert out_seq \rvert$: $(\forall j. 0 \leq j < \lvert out_seq[i] \rvert$: $out_seq[i][j] = Output[i][j]))$

Table 4.1: List of all properties in the summed-area table algorithm. Even numbers indicate the relation between the ghost and concrete variables. Odd numbers shows the relation between the ghost variables before and after each phase of the algorithm.

algorithm over the sequences. From the postcondition of the *transpose* function we have **Properties 4.7** and **4.8** (Table 4.1) in line 11 of Algorithm 4.2. **Property 4.7** shows that *tmp2_seq* is the transposition of *tmp1_seq* and **Property 4.8** relates the ghost variable, *tmp2_seq* to the actual array *Temp*.

Then we have the second prefix sum function to compute the inclusive prefix sum of elements in *Temp* and store it in *Output*. We define a ghost variable, *tmp3_seq*, to store the inclusive prefix sum of elements in *tmp2_seq*, which is the same as *Temp* (lines 14 in Algorithm 4.2). Figure 4.11: step 3, shows this phase against the ghost variables. From the postcondition of the verified inclusive prefix sum (see the previous chapter), it follows that **Properties 4.9** and **4.10** (Table 4.1) hold in line 15 of Algorithm 4.2. **Property 4.9** specifies that *tmp3_seq* contains the inclusive prefix sum of the elements in *tmp2_seq*. **Property 4.10** shows the relation between *tmp3_seq* and *Output*.

The last phase of the algorithm is the second transposition, but this time for *Output*. Therefore, we now apply the transposition function to the *tmp3_seq* ghost variable and store the result in another ghost variable, *out_seq* in line 20 of Algorithm 4.2 (see Figure 4.11: step 4). At this point (line 21 in Algorithm 4.2) we have **Properties 4.11** and **4.12** (Table 4.1). **Property 4.11** indicates that *out_seq* is the transposition of *tmp3_seq* and **Property 4.12** relates *out_seq* to *Output*.

As we can see in **Property 4.12**, we relate the final result between the ghost variable, *out_seq* and the actual array *Output*, but we still should reason about the values in *out_seq* (and correspondingly *Output*). To accomplish this, we prove several corollaries following from the properties in Table 4.1:

Corollary 4.1 From **Properties 4.11** and **4.9** we have:

$$(\forall i.0 \le i < |out_seq|\colon (\forall j.0 \le j < |out_seq[i]|\colon\ out_seq[i][j] = \sum_{t=0}^{i} tmp2_seq[j][t])).$$

Corollary 4.2 From **Corollary 4.1** and **Property 4.7** we have:

$$(\forall i.0 \le i < |out_seq|\colon (\forall j.0 \le j < |out_seq[i]|\colon\ out_seq[i][j] = \sum_{t=0}^{i} tmp1_seq[t][j])).$$

Corollary 4.3 From **Corollary 4.2** and **Property 4.5** we have:

$$(\forall i.0 \le i < |out_seq|\colon (\forall j.0 \le j < |out_seq[i]|\colon$$
$$out_seq[i][j] = \sum_{t_1=0}^{i} \sum_{t_2=0}^{j} inp_seq[t_1][t_2])).$$

Corollary 4.1 relates *out_seq* to *tmp2_seq* (Figure 4.11: step 5). Corollary 4.2 shows the relation between the ghost variables *out_seq* and *tmp1_seq* (Figure 4.11: step 6). Corollary 4.3 proves the relation between *inp_seq* and *out_seq* (Figure 4.11: step 7). As *inp_seq* and *out_seq* are the same as *Input* and *Output* (**Property 4.12**), respectively, we conclude functional correctness.

In this verification, we could easily reuse the specification of the verified prefix sum algorithm in a straightforward way without proving more properties. As a consequence, the verification of the summed-area table algorithm takes much less time than the verification of the prefix sum algorithm. Moreover, we verified the parallel transposition, which is also a primitive operation in GPGPUs, and thus its verification can be reused for the verification of other algorithms that use this operation.

4.4 Towards CUDA Verification

In the previous chapter we discussed how we developed the verification of the prefix sum algorithm for its CUDA implementation. We also could successfully verify the CUDA version of the stream compaction algorithm. However, we are not able to verify the CUDA implementation of summed-area table. The reason is that we should extend CUDA support in VerCors for two dimensional thread blocks, which is currently not supported. Therefore, it remains as future work to verify the CUDA version of this algorithm.

Since the CUDA implementation of stream compaction reuses the prefix sum implementation, the complexity of the annotated CUDA file increases. As a result, we encounter non-termination problems with the verification in the tool, as Z3 cannot prove nor refute the properties when they are all in one file. To solve this problem, we had to split up the code over several files. First of all, we removed the host part and only preserved the kernel part. This does not make any difference for the verification as there is only one parallel block and the algorithm completely runs inside the kernel. Second, to keep the verification effort manageable, we prove all lemmas in a separate file and only use their specification during the kernel verification. Again, this does not affect what is proven, but it does have a significant impact on the verification time. It is future work to engineer some support to do this splitting automatically.

4.5 Related Work

There is no previous work on formally verifying the summed-area table algorithm on GPUs. The closest related work to our work is by Chong et al. [CDK$^+$13] who they verify data-race freedom of the parallel stream compaction algorithm. They first extend the two-thread reduction technique by proposing barrier invariants in order to reason about data-dependent GPU programs. In data-dependent GPU programs, data or control flow depends on the input of the (GPU) program. They then select stream compaction as a good example of a data-dependent program. Using their abstraction technique, they augment barriers with invariants in or-

der to preserve properties of shared states across the barriers in GPU programs. They implement their approach in GPUVerify. Similar to our approach they use Blelloch's prefix sum algorithm in the stream compaction algorithm. The barrier invariants that they specify in the up-sweep and down-sweep-phases (of Blelloch's algorithm) are different than in our work. Their invariants are per element in the array. That means, for each location in the array, they specify what value it has, according to each iteration of the algorithm in both phases. In our opinion, the two invariants that we provide in Blelloch's algorithm are simpler to specify as they consider how the values change in each iteration and what is the relation of those (new) values in the corresponding iterations of both phases.

4.6 Conclusion

In this chapter, we have proven data-race freedom and functional correctness of the parallel stream compaction and summed-area table algorithms, for an arbitrary input size by encoding the algorithms into the VerCors verifier. The two algorithms are widely used as primitive operations in other algorithms (e.g., collision detection and sparse matrix compression). Proving functional correctness of both algorithms is challenging because both use other parallel algorithms as sub-routine (e.g., prefix sum and transposition). To overcome these challenges, we reuse previous work on the verification of parallel prefix sum algorithms. It is straightforward to reuse the verification of prefix sum in the summed-area table algorithm. However, the transposition operation is used intermittently in addition to the prefix sum sub-routine. We should establish a formal relation between each of these intermediate steps in order to reason about the final result. In the stream compaction algorithm, since the input to the prefix sum sub-routine is a flag array, we should prove more properties of the prefix sum. Moreover, we define ghost variables and suitable functions that mimic the actual computations in the algorithms.

The complete verification of both algorithms took 2 weeks, whereas in comparison the verification of prefix sum took a couple of months. This shows that less effort is needed to verify complicated algorithms by reusing verified sub-routines in practice. Therefore, we believe that now it will be a minimal effort to verify even more complex algorithms that are built on top of the stream compaction and summed-area table algorithms. As future work, we would like to investigate how a substantial part of the required annotations, in particular those related to permissions, can be generated automatically. In addition, based on our verifications, we plan to develop a library of general properties for common GPGPU sub-routines in VerCors.

CHAPTER 5

Verification of Parallel Bellman–Ford Single-Source Shortest Path Algorithm

In the previous chapters, we have seen how to verify parallel algorithms to solve prefix sums, stream compaction and summed-area table problems. In addition to these problems, also finding shortest distance between two points is a fundamental problem. In general, many real-world problems such as routing are actually graph problems. To develop efficient solutions to such problems, more and more parallel graph algorithms are proposed. Thus, they are good candidates to be parallelized on GPUs. As a result, they are also a good case study from the verification point of view.

Therefore, this chapter discusses the mechanized verification of a commonly used parallel graph algorithm, namely the Bellman–Ford algorithm, which provides an inherently parallel solution to the Single-Source Shortest Path problem. Concretely, we verify an unoptimized GPU version of the Bellman–Ford algorithm, using the VerCors verifier. The main challenge that we had to address was to find suitable global invariants of the graph-based properties for mechanized verification. The verification of this algorithm provides the basis to verify other, optimized implementations of the algorithm. Moreover, it may also provide a good starting point to verify other parallel graph-based algorithms.

5.1 Introduction

Graph algorithms play an important role in computer science, as many real-world problems can be handled by defining suitable graph representations. This makes the correctness of such algorithms crucially important. As the real-world problems that we represent using graphs are growing exponentially in size—think for example about internet routing solutions—we need highly efficient, but still correct(!),

graph algorithms. Massively parallel implementation of graph algorithms on GPUs can help to obtain the required efficiency, but also introduces extra challenges to reason about the correctness of such parallel graph algorithms.

This chapter uses deductive verification, based on permission-based separation logic, to develop a mechanized proof of a parallel GPU-based graph algorithm. The concrete graph algorithm that we study here is the Bellman–Ford algorithm [FJ56, Bel58], a solution for the Single-Source Shortest Path (SSSP) problem. This algorithm computes the shortest distance from a specific vertex to all other vertices in a graph, where the distance is measured in terms of arc weights. Other solutions exist for this problem, such as Dijkstra's shortest path algorithm [Dij59]. However, the Bellman–Ford algorithm is inherently parallel, which makes it suitable to be used on massively parallel architectures, such as GPUs.

In this chapter, we prove data-race freedom and functional correctness of a standard parallel GPU-based Bellman–Ford algorithm. This correctness proof can be used as a starting point to also derive correctness of the various optimized implementations that have been proposed in the literature [BB16, AD15, HP14, SS17, JUK+14, SE17]. Moreover, this verification and the experiences with automated reasoning about GPU-based graph algorithms will also provide a good starting point to verify other parallel GPU-based graph algorithms.

The main challenge that we had to address in this verification, was to find the suitable global invariants to reason about the graph-based algorithm, and in particular to make those amenable to mechanized verification. As mentioned before, the Bellman–Ford algorithm is inherently parallel, and our proof indeed demonstrates this.

In the rest of the chapter, we first explain the SSSP problem and how the Bellman–Ford algorithm is a parallel solution to the problem. Next, we outline the manual correctness proof, and then we discuss how we formalize this proof in VerCors.

5.2 SSSP and the Bellman–Ford Algorithm

This section explains the single-source shortest path problem and the Bellman–Ford Algorithm.

SSSP A directed weighted graph $G = (V, A, w)$ is a triple consisting of a finite set V of vertices, an binary arc relation $A \subseteq V \times V$, and a (non-negative) weight function $w : A \to \mathbb{N}$ over arcs. In the remainder of this chapter we assume that A is irreflexive, since we do not consider graphs that contain self-loops. The *source* and *destination* of any arc $a = (u, v) \in A$ is defined $src(a) \triangleq u$ and $dst(a) \triangleq v$,

Algorithm 5.1 Sequential Bellman-Ford's Algorithm

```
1: function BELLMAN-FORD(G, s, cost)
2:     cost[s] = 0, cost[V \ {s}] = ∞; // Initialization
3:     for (i = 0 up to |V| − 2) do // Start |V| − 1 rounds of cost relaxations
4:         for (every a ∈ A) do // Check every arc for potential cost relaxations
5:             if  cost[src[a]] + w[a] < cost[dst[a]] then
6:                 cost[dst[a]] = cost[src[a]] + w[a]; // Perform a cost relaxation
```

respectively. Any finite arc sequence $P = (a_0, a_1, \ldots, a_n) \in A^*$ is a *path in G* if $dst(a_i) = src(a_{i+1})$ for every $0 \le i < n$. We say that P is an (u, v)*-path* if $src(a_0) = u$ and $dst(a_n) = v$, under the condition that $n \ge 0$. The *length of P* is denoted $|P|$ and defined to be $n + 1$. The *weight* of any path P is denoted $w(P)$, with w overloaded and lifted to sequences of arcs $A^* \to \mathbb{N}$ as follows: $w(a_0, a_1, \ldots, a_n) \triangleq \Sigma_{i=0}^{n} w(a_i)$. Any path P is *simple* if all its vertices are unique. Finally, any (u, v)-path P is a *shortest* (u, v)*-path in G* if for every (u, v)-path Q in G it holds that $w(P) \le w(Q)$.

Bellman–Ford The Bellman–Ford algorithm [FJ56, Bel58] solves the *Single-Source Shortest Path (SSSP) problem*: given any input graph $G = (V, A, w)$ and vertex $s \in V$, find for any vertex $v \in V$ reachable from s the weight of a shortest (s, v)-path. Algorithm 5.1 shows the pseudo-code of Bellman–Ford[1]. It takes as input a graph G , starting vertex s and a *cost* array. The idea is to associate a *cost* to every vertex v, which amounts to the weight of a shortest (s, v)-path that has been found up to round i of the algorithm. Initially, s has cost 0, while all other vertices start with cost ∞ (line 2). Then the algorithm operates in $|V| - 1$ rounds (line 3). In every round i, the cost of vertex v is *relaxed* on lines 5-6 in case a cheaper possibility of reaching v is found. After $|V| - 1$ such rounds of relaxations, the weights of the shortest paths to all reachable vertices have been found, intuitively because no simple path can contain more than $|V| - 1$ arcs.

The Bellman–Ford algorithm can straightforwardly be parallelized on a GPU, by executing the iterations of the for-loop on line 4 in parallel, thereby exploiting that arcs can be iterated over in arbitrary order. Such a parallelization requires lines 5-6 to be executed atomically, and all threads to synchronize (by a possibly implicit barrier) between every iteration of the round loop. Algorithms 5.2 and 5.3 show the pseudo-code of CPU host and GPU kernel, respectively[2].

This chapter demonstrates how VerCors is used to mechanically verify *sound-*

[1] In the pseudo-code vertices and arcs are assumed integers (counted from 0).

[2] atomicMin() is a built-in GPU function that compares its two arguments and assigns the minimum one to the first argument.

Algorithm 5.2 Host (CPU)

```
1: function BELLMAN-FORD(G, s, cost)
2:     cost[s] = 0, cost[V \ {s}] = ∞;
3:     for (i = 0 up to |V| − 2) do
4:         Launching Bellman-Ford Kernel (with A threads)
```

Algorithm 5.3 Device (GPU) Bellman-Ford Kernel

```
1: kernel BELLMAN-FORD-KERNEL(G, s, cost)
2:     atomicMin( &cost[dst[tid]], cost[src[tid]] + w[tid] )
```

ness and *completeness* of the parallelized version of the Bellman–Ford algorithm. Soundness in this context means that, after completion of the algorithm, for any $v \in V$ such that $cost[v] < \infty$ it holds that there exists a shortest (s, v)-path P such that $cost[v] = w(P)$. The property of completeness is that, for any v if there exists an (s, v)-path P after completion of the algorithm, it holds that $cost[v] < \infty$. In addition to soundness and completeness we also use VerCors to verify data-race freedom of parallel Bellman–Ford.

5.3 Approach

Our strategy for verifying parallel Bellman–Ford is to first construct an informal pen-and-paper proof of its correctness, and then to encode this proof in VerCors to mechanically check all proof steps. This section elaborates on the (informal) correctness argument of Bellman–Ford, after which the next section explains how this argument is encoded in, and then confirmed by, VerCors.

Postconditions Proving correctness of parallel Bellman–Ford amounts to proving that the following three postconditions hold after termination of the algorithm, when given as input a graph $G = (V, A, w)$, starting vertex $s \in V$ and a cost array:

$$\forall v \,.\, cost[v] < \infty \Rightarrow \exists P \,.\, Path(P, s, v) \wedge w(P) = cost[v] \tag{PC1}$$

$$\forall v \,.\, (\exists P \,.\, Path(P, s, v)) \Rightarrow cost[v] < \infty \tag{PC2}$$

$$\forall v \,.\, cost[v] < \infty \Rightarrow \forall P \,.\, Path(P, s, v) \Rightarrow cost[v] \leq w(P) \tag{PC3}$$

The predicate $Path(P, u, v)$ expresses that P is an (u, v)-path in G.

These three postconditions together express that *cost* characterizes reachable states as well as shortest paths. PC1 and PC2 imply soundness and completeness of

reachability: $cost[v] < \infty$ if and only if v is reachable from s. PC3 additionally ensures that *cost* contains the weights of all shortest paths in G.

Invariants Our approach for proving the three postconditions above is to introduce *round invariants*: invariants for the loop on line 3 in Algorithm 5.1 that should hold at the start and end of every round i for each thread. The proposed (round) invariants are:

$$\forall v \,.\, cost[v] < \infty \Rightarrow \exists P \,.\, Path(P, s, v) \wedge w(P) = cost[v] \tag{INV1}$$

$$\forall v \,.\, (\exists P \,.\, Path(P, s, v) \wedge |P| \leq i) \Rightarrow cost[v] < \infty \tag{INV2}$$

$$\forall v \,.\, cost[v] < \infty \Rightarrow \forall P \,.\, Path(P, s, v) \wedge |P| \leq i \Rightarrow cost[v] \leq w(P) \tag{INV3}$$

One can prove that the round invariants imply the postconditions after termination of the round loop, as then $i = |V| - 1$. PC1 immediately follows from INV1 without additional proof. Proving that PC2 and PC3 follow from INV2 and INV3 respectively requires more work since these postconditions quantify over paths of unbounded length.

Therefore, we introduce an operation $simple(P)$ that removes all cycles from any given (u, v)-path P, and gives a simple (u, v)-path. The three main properties of *simple* that are needed for proving the postconditions are $|simple(P)| \leq |V| - 1$, $|simple(P)| \leq |P|$, and $w(simple(P)) \leq w(P)$ for any P. The latter two properties hold since $simple(P)$ can only shorten P. This auxiliary operation makes it easy to establish the postconditions. Here we detail the proof for PC2 and PC3:

Lemma 5.1 If $i = |V| - 1$ then INV2 implies PC2.

Proof. Let v be an arbitrary vertex and P be an arbitrary (s, v)-path. We know that $|simple(P)| \leq |V| - 1$, and the existence of this path already completes the proof due to INV1 . □

Lemma 5.2 If $i = |V| - 1$ then INV3 implies PC3.

Proof. Let v be an arbitrary vertex such that $cost[v] < \infty$, and P be an arbitrary (s, v)-path. Then $cost[v] \leq w(P)$ is shown by instantiating INV3 with v and $simple(P)$, from which one can easily prove $cost[v] \leq w(simple(P)) \leq w(P)$. □

Preservation of invariants. However, proving that each round of the algorithm preserves the round invariants is significantly more challenging. It is non-trivial to show that validity of invariants INV1–INV3 at round $i + 1$ follows from their

validity at round i combined with the contributions of all threads in round i. An additional difficulty is that cost relaxations are performed in arbitrary order.

Our approach is to first work out the proof details in pen-and-paper style, and to later encode all proof steps in VerCors. We discuss the proofs for preservation of each invariant:

Lemma 5.3 Every iteration of the loop on lines 3-6 in Algorithm 5.1 preserves INV1.

Proof. The proof of this invariant is quite easy. Assume that the invariant holds in iteration i. Given any vertex v, if $cost[v]$ has changed in iteration $i+1$, then a witness path is available, and otherwise the invariant still holds by assumption in iteration $i+1$. □

Lemma 5.4 Every iteration of the loop on lines 3-6 in Algorithm 5.1 preserves INV2.

Proof. Suppose that INV1–INV3 hold on round i, that $i < |V|-1$, and that all cost relaxations have happened for round i (i.e., lines 4-6 have been fully executed). We show that INV2 holds for $i+1$. We write $\mathsf{old}(cost[v])$ to refer to the "old" cost that any v had at the beginning of round i. We create a proof by contradiction. Let v be an arbitrary vertex and P be an arbitrary (s, v)-path such that $|P| \leq i+1$, and $cost[v] = \infty$. We show that such a vertex v cannot exist. Since $cost[v] = \infty$, we know that $v \neq s$ and therefore $|P| > 0$ (i.e., P consists of at least one arc). Let a be the last arc on P so that $dst(a) = v$, and let P' be the path P but without a, so that P' is an $(s, src(a))$-path such that $|P'| \leq i$. Let us abbreviate $src(a)$ as v'. It follows from invariant INV2 that $\mathsf{old}(cost[v']) < \infty$. Then, it must be that $\mathsf{old}(cost[v]) < \infty$, since otherwise the thread responsible for the arc a would have updated (relaxed) $cost[v]$ to be $\mathsf{old}(cost[v']) + w(a)$, which is not ∞. But this is impossible, since that would mean that the responsible thread updated $cost[v]$ to be ∞ in round $i+1$, while it was a concrete natural number before, in round i. Therefore, we conclude that such a vertex v cannot exist. □

Lemma 5.5 Every iteration of the loop on lines 3-6 in Algorithm 5.1 preserves INV3.

Proof. Suppose again that INV1–INV3 hold on round i, that $i < |V| - 1$, and that all cost relaxations have happened for round i (i.e., lines 4-6 have been fully executed). We show that INV3 holds for $i+1$ by contradiction.

Suppose that there exists a vertex v and an (s, v)-path P such that $cost[v] < \infty$, $|P| \leq i + 1$, and $w(P) < cost[v]$. It must be the case that $|P| = i + 1$, since otherwise, if $|P| < i + 1$ were the case, then INV1 and INV2 together would imply that $\mathsf{old}(cost[v]) < w(P)$, which is impossible since vertex costs can only decrease. Therefore P consists of at least one arc. Let a be the last arc on P so that $dst(a) = v$, and let P' be the path P but without a, so that P' is an $(s, src(a))$-path of length i. Let us abbreviate $src(a)$ as v'. Instantiating INV2 and INV3 with v' and P' gives $\mathsf{old}(cost[v']) < \infty$ and $\mathsf{old}(cost[v']) < w(P')$.

Let us now consider what a's thread could have done in round i. When this thread got scheduled it must have observed that v' and v had some intermediate costs, which we refer to as $obs_{v'}$ and obs_v, respectively, for which it holds that $cost[v'] \leq obs_{v'} \leq \mathsf{old}(cost[v'])$ and $cost[v] \leq obs_v \leq \mathsf{old}(cost[v])$. And since

$$obs_{v'} + w(a) \leq \mathsf{old}(cost[v']) + w(a) \leq w(P') + w(a) = w(P) < cost[v] \leq obs_v$$

we know that a's thread must have updated the cost of v to be $obs_{v'} + w(a)$ in its turn. Since v's cost might have decreased further in round i by other threads, we have $cost[v] \leq obs_{v'} + w(a) \leq w(P)$, which contradicts $w(P) < cost[v]$. □

The proof outlines emphasize the non-triviality of verifying the Bellman–Ford algorithm using mechanized code verifiers. In the next section, we explain how to automate this process in the VerCors verifier.

5.4 Proof Mechanization

So far the correctness argument of parallel Bellman–Ford has been presented at the abstract level of mathematical definitions and pseudocode. This section discusses how this abstract reasoning translates to the parallel GPU-based version of Bellman–Ford, by formalizing its correctness proof in VerCors. This requires (i) encoding all specifications introduced in Section 5.3 into the VerCors specification language, (ii) adding additional permission specifications to guarantee data-race freedom, and (iii) using these specifications to formulate pre- and postconditions, as well as loop and atomic block invariants for the algorithm encoding. For step (i), we should consider GPU memory (as a scarce resource) that imposes more restrictions on how to represent large graphs in an efficient way (e.g., using a one-dimensional array instead of matrices, and assigning threads to arcs instead of vertices). For step (iii), the main challenge was to encode the lemmas and their proofs, as introduced in Section 5.3. We use *lemma functions* for this, which are pure functions whose function specification corresponds to the lemma property. The challenge was to encode the proofs of these lemmas in VerCors, as discussed in more detail below. The end result of our verification effort is a

```
1  /*@ context Graph(V, A, src, dst, w);
2      context (\forall* int i; i ≥ 0 && i < A;
3               Perm(src[i], read) ** Perm(dst[i], read) ** Perm(w[i], read));
4      context (\forall* int i; i ≥ 0 && i < V; Perm(cost[i], write));
5      ens (\forall int v; v ≥ 0 && v < V && cost[v] < inf(); // Encoding of PC1
6               (\exists seq<int> P; Path(V, A, src, dst, w, s, v, P);
7                 Weight(V, A, src, dst, w, P) == cost[v]));
8      ens (\forall int v; v ≥ 0 && v < V && // Encoding of PC2
9               (\exists seq<int> P; Path(V, A, src, dst, w, s, v, P); true);
10                cost[v] < inf());
11     ens (\forall int v; v ≥ 0 && v < V && cost[v] < inf(); // Encoding of PC3
12              (\forall seq<int> P; Path(V, A, src, dst, w, s, v, P);
13                cost[v] ≤ Weight(V, A, src, dst, w, P))); @*/
14 void Host_BF(int V, int A, int[] src, int[] dst, int[] w, int[] cost){
15   /*@ inv (\forall* int i; i ≥ 0 && i < A;
16              Perm(src[i], read) ** Perm(dst[i], read) ** Perm(w[i], read));
17       inv (\forall* int i; i ≥ 0 && i < V; Perm(cost[i], write));
18       inv (\forall int v; v ≥ 0 && v < V && cost[v] < inf(); // Encoding of INV1
19             (\exists seq<int> P; Path(V, A, src, dst, w, s, v, P);
20              Weight(V, A, src, dst, w, P) == cost[v]));
21       inv (\forall int v; v ≥ 0 && v < V && // Encoding of INV2
22             (\exists seq<int> P; Path(V, A, src, dst, w, s, v, P); |P| ≤ i);
23              cost[v] < inf());
24       inv (\forall int v; v ≥ 0 && v < V && cost[v] < inf(); // Encoding of INV3
25             (\forall seq<int> P; Path(V, A, src, dst, w, s, v, P) && |P| ≤ i;
26              cost[v] ≤ Weight(V, A, src, dst, w, P)));@*/
27   for(i = 0; i < V-1; i++){
28       par Kernel_BF(int tid = 0..A){
29         /*@ inv Perm(...) // Permission
30             inv ... // Encoding of INV1, INV2 and INV3 @*/
31         atomic{
32           if (cost[src[tid]] + w[tid] < cost[dst[tid]])
33               cost[dst[tid]] = cost[src[tid]] + w[tid];
34         }
35       }
36       lemma_inv_pres(...); // Applying invariant preservation lemma functions
37   }
38   lemma_post_establ(...); // Applying postcondition establishment lemma functions
39 }
```

Figure 5.1: Verification of the Bellman–Ford algorithm in VerCors.

machine-checked proof of a GPU version of parallel Bellman-Ford. The remainder of this section elaborates on the formalization of the informal specifications and proof outlines in VerCors, and on how these are used to verify the GPU-based parallel Bellman-Ford.

```
1 pure bool Graph(int V, int A, int[] src, int[] dst, int[] w) =
2   V > 0 && A > 0 && |src| == A && |dst| == A && |w| == A &&
3   (\forall int i; i ≥ 0 && i < A;
4    src[i] ≥ 0 && src[i] < V && dst[i] ≥ 0 && dst[i] < V &&
5    src[i] != dst[i] && w[i] > 0 &&
6    (\forall int j; j ≥ 0 && j < A && i != j && src[i] == src[j]; dst[i] != dst[j]));
```

Figure 5.2: The *Graph* predicate, that determines whether *src*, *dst* and *w* form a graph.

```
1 /*@ req Graph(V, A, src, dst, w);
2     ens (\forall int i; 0 ≤ i && i < A;
3              Path(V, A, src, dst, w, src[i], dst[i], int[] {i})); @*/
4 pure bool Path(int V, int A, int[] src, int[] dst, int[] w, int u, int v, int[]
      P) = u ≥ 0 && u < V && v ≥ 0 && v < V &&
5   (\forall int i; i ≥ 0 && i < |P|; P[i] ≥ 0 && P[i] < A) &&
6   (|P| == 0 ==> u == v) && (|P| > 0 ==> src[P[0]] == u && dst[P[|P|-1]] == v) &&
7   (\forall int i; i ≥ 0 && i < |P|-1; dst[P[i]] == src[P[i+1]]);
```

Figure 5.3: The *Path* predicate to representation a path in *Graph*.

Proof outline and specification encoding. Figure 5.1 presents a simplified overview of our specification of parallel Bellman–Ford. Lines 28-35 show the annotated code runs on GPU (kernel) and the rest is the CPU part (host). Observe that the algorithm uses a representation of directed weighted graphs that is typical for GPU implementations: using three arrays, *src*, *dst* and *w*.

On line 1 in the specification we require (and ensure)[3] that these three arrays indeed form a graph, by means of the predicate $Graph(V, A, src, dst, w)$, as defined in Figure 5.2. The integer V represents the total number of vertices, and A the total number of arcs. Then any index $a \in [0, A)$ represents an arc from $src[a]$ to $dst[a]$ with weight $w[a]$. Similarly, any index $v \in [0, V)$ represents a vertex in the graph such that $cost[v]$ is the current cost assigned to v by the algorithm. The integer s, with $0 \leq s < V$, is the starting vertex. This representation can handle large graphs on GPU memory, and by assigning threads to arcs more parallelism and hence more performance can be obtained.

Lines 5–13 and 18–26 contain the VerCors encoding of the postconditions and round invariants introduced in Section 5.3, respectively. These encodings are defined over various other predicates such as *Path* and *Weight*, whose definitions are

[3] The keyword `context` is an abbreviation for both `requires` and `ensures`.

```
1 /*@ req Graph(V, A, src, dst, w);
2     req (\forall int i; i ≥ 0 && i < |P|; P[i] ≥ 0 && P[i] < A)
3     ens \result ≥ 0 @*/
4 pure int Weight(int V, int A, int[] src, int[] dst, int[] w, int[] P) =
5    |P| > 0 ? w[P[0]] + Weight(V, A, src, dst, P[1..]) : 0;
```

Figure 5.4: The *Weight* predicate of any path.

defined in Figure 5.3 and Figure 5.4.

Data-race freedom. Verifying data-race freedom requires explicitly specifying ownership over heap locations using fractional permissions. Lines 2-4 and 15-17 indicate that initially and in each iteration of the algorithm we have read permission over all locations in *src*, *dst* and *w*. Moreover, we also have write permission over all locations in *cost*. Within the parallel block (kernel), threads execute in parallel, meaning that the updates to *cost* have to be done atomically (lines 31-34). The atomic invariants specify shared resources and properties that may be used by a thread while in the critical section (lines 29-30). After leaving the critical section, the thread should ensure all the atomic invariants are re-established.

The atomic invariant on line 29 specifies that each thread within the critical section has read permission over all locations in *src*, *dst* and *w* and write permission in *cost*. Note that the atomic block execute in an arbitrary order, but as there always is at most one thread within the critical section, this is sufficient to guarantee data-race freedom.

Lemma functions. As mentioned above, to show the preservation of INV1, INV2 and INV3 we apply the corresponding lemmas at the end of the loop (line 36). Note that these invariants must hold in the kernel as well (line 30). Similarly, to establish PC1, PC2 and PC3 we apply the corresponding lemmas after termination of the loop when $i = |V| - 1$ (line 38).

All the proofs of the lemmas mentioned in Section 5.3 that show the preservation of the round invariants and establishment of postconditions are encoded in VerCors as *lemma functions* [VME18, GT16]. Lemma functions have specifications that capture the desired property, while the proof is encoded as a side effect-free imperative program. Most of our lemmas (e.g., the proof of Lemma 5.5) were proven by contradiction. Proving a property ϕ by contradiction amounts to proving $\neg\phi \Rightarrow \mathsf{false}$. Therefore, to show preservation of, e.g., INV3 (Lemma 5.5), we

```
1  /*@ req Graph(V, A, src, dst, w) && i ≥ 0 && i < V-1 && |P| ≤ i+1;
2      req cost[v] ≤ oldcost[v] && oldcost[v] < inf();
3      req Weight(V, A, src, dst, w, P) < cost[v];
4      req (\forall int a; a >= 0 && a < A &&
5       oldcost[src[a]]+w[a] < oldcost[dst[a]]; cost[dst[a]] ≤ oldcost[src[a]]+w[a]);
6      req (\forall int v; v ≥ 0 && v < V && cost[v] < inf(); // Encoding of INV1
7               (\exists seq<int> P; Path(V, A, src, dst, w, s, v, P);
8                  Weight(V, A, src, dst, w, P) == oldcost[v]));
9      req (\forall int v; v ≥ 0 && v < V && // Encoding of INV2
10              (\exists seq<int> P; Path(V, A, src, dst, w, s, v, P) && |P| ≤ i;
11                 oldcost[v] < inf();
12     req (\forall int v; v ≥ 0 && v < V && oldcost[v] < inf(); // Encoding of INV3
13              (\forall seq<int> P; Path(V, A, src, dst, w, s, v, P) && |P| ≤ i;
14                 oldcost[v] ≤ Weight(V, A, src, dst, w, s, v, P)));
15     ens false; @*/
16 pure bool lemma_5(int i, int v, int[] P, ...) {
17    //@ assert |P| > 0 && |P| == i+1;
18    //@ ghost int a = P[|P|-1]; int[] P′ = P[0..|P|-2];
19    //@ assert |P′| ≤ i;
20    //@ ghost int v′ = src[a];
21    //@ assert oldcost[v′] ≤ Weight(V, A, src, dst, w, P′); // From INV2 and INV3
22    //@ assert cost[v′] ≤ oldcost[v′] && cost[v′] ≤ Weight(V, A, src, dst, w, P′);
23    //@ assert lemma_transitivity(V, A, src, dst, w, s, v′, v, P′, int[] {a});
24    //@ assert cost[v] ≤ Weight(V, A, src, dst, w, P); // Contradiction
25 }
```

Figure 5.5: The (simplified) VerCors encoding of Lemma 5.5.

proved that $(\text{INV1}(i) \wedge \text{INV2}(i) \wedge \text{INV3}(i) \wedge \phi(i) \wedge \neg\text{INV3}(i+1)) \Rightarrow \mathsf{false}$, with $\phi(i)$ describing the contributions of all threads in round i.

Figure 5.5 shows how the VerCors encoding of Lemma 5.5 looks, where the lemma is implicitly quantified over the function parameters: iteration round i, vertex v, and path P. The function body encodes all proof steps. The main challenge was finding the precise assertions that explicitly describe all the steps from the informal proof. In particular, we had to prove various auxiliary lemmas such as *lemma_transitivity* (line 23), which models the transitivity property of paths along with its weight, and which required an induction over paths in its proof. Figure 5.6 shows how to prove the transitivity property.

Similarly, we prove Lemma 5.4 by contradiction in VerCors as shown in Figure 5.7. Note that the proofs are for an arbitrary vertex and path within the domain of the universal quantified variables.

```
1  /*@ req Graph(V, A, src, dst, w);
2      req Path(V, A, src, dst, w, u, t, P);
3      req Path(V, A, src, dst, w, t, v, Q);
4      ens Path(V, A, src, dst, w, u, v, P + Q);
5      ens Weight(V, A, src, dst, w, P + Q) == Weight(V, A, src, dst, w, P) +
6                                              Weight(V, A, src, dst, w, Q) @*/
7  pure bool lemma_transitivity(int V, int A, int[] src, int[] dst, int[] w, int u,
       int t, int v, int[] P, int[] Q) =
8    |P| > 0 =>
9    (
10     lemma_transitivity(V, A, src, dst, w, dst[P[0]], t, v, P[1..], Q) &&
11     Path(V, A, src, dst, w, src[P[0]], v, P[0] + P[1..0] + Q) &&
12     (P[0] + P[1..]) + Q == P[0] + (P[1..] + Q) &&
13     P[0] + P[1..] == P
14     (P + Q)[0] == P[0] &&
15     (P + Q)[1..] == P[1..] + Q
16   );
```

Figure 5.6: Transitivity property of paths.

5.5 Evaluation and Discussion

The algorithm encoding and its specification consists of 541 lines of code. Of these 541 lines, 30 are for the encoding of the algorithm (5.5%) and the remaining 511 are specification (94.5%). The specification part can be subdivided further: of the 511 lines, 6.1% is related to permissions, 30.7% to invariant preservation proofs, 45.1% to proofs for establishing the postconditions, and 18.1% to definitions (e.g., of graphs and paths) and proving basic properties.

The total verification effort was about six weeks. Most of this time was spent on the mechanization aspects: spelling out all the details that were left implicit in the pen-and-paper proof. The fully annotated Bellman–Ford implementation takes about 12 minutes to verify using VerCors on a Macbook Pro (early 2017) with 16GB RAM, and an Intel Core i5 3.1GHz CPU.

This case study confirms the importance of lemma functions in verifying non-trivial case studies, and in particular for encoding proofs by contradiction, which are common in the context of graphs. Moreover, (recursive) lemma functions are important for proofs by induction, since SMT solvers does not provide any kind of proof by induction. This chapter also gives a representation of graphs that is suitable for GPU architectures, and can form the foundation of other verifications. Finally, we learned that deductive code verifiers are powerful enough to reason about non-trivial parallel algorithms—but they cannot do this yet without the

```
1  /*@ req Graph(V, A, src, dst, w) && i ≥ 0 && i < V-1 && |P| ≤ i+1;
2      req cost[v] ≤ oldcost[v] && oldcost[v] < inf();
3      req cost[v] == inf() && cost[src] == 0 && oldcost[s] == 0;
4      req (\forall int a; a >= 0 && a < A &&
5       oldcost[src[a]]+w[a] < oldcost[dst[a]]; cost[dst[a]] ≤ oldcost[src[a]]+w[a]);
6      req (\forall int v; v ≥ 0 && v < V && // Encoding of INV2
7               (\exists seq<int> P; Path(V, A, src, dst, w, s, v, P) && |P| ≤ i;
8                   oldcost[v] < inf();
9      req (\forall int a; a ≥ 0 && a < A && oldcost[src[a]] < inf();
10              cost[dst[a]] < inf() );
11     ens false; @*/
12 pure bool lemma_4(int i, int v, int[] P, ...) {
13     //@ assert |P| ≤ i+1;
14     //@ assert cost[v] == inf() && cost[src] == 0;
15     //@ assert v != src && |P| > 0;
16     //@ ghost int a = P[|P|-1]; int[] P' = P[0..|P|-2];
17     //@ assert |P'| ≤ i;
18     //@ ghost int v' = src[a];
19     //@ assert oldcost[v'] < inf(); // From INV2
20     //@ assert cost[v] < inf(); // Contradiction
21 }
```

Figure 5.7: The (simplified) VerCors encoding of Lemma 5.4.

human expertize to guide the prover.

5.6 Related Work

There are many different implementations of shortest path algorithms on GPUs. Harish et al. [HN07, HVN09] were the first to implement graph algorithms, and in particular shortest path algorithms on GPUs. There are multiple works that improve Harish's method [KMT11, OIH12, BB16, SE17]. The goal of those works is to improve performance by proposing new optimization techniques on GPUs, and also taking advantage of new features in GPU architecture and programming. There is also some work to develop frameworks to provide graph processing engines on GPUs [ZH14, KVGB14, WDP+16]. The goal of those works is to provide an abstraction to ease implementing graph algorithms on GPUs. Pai et al. [PP16] propose an intermediate program representation, called IrGL, to describe graph algorithms. Then a compiler can automatically generate optimized CUDA code from IrGL. They claim that the performance of the generated code is comparable and for some cases even faster than hand-written CUDA code. They use 8 graph algorithms to evaluate their approach.

There are some works on the verification of graph algorithms. The work that is closest to ours is by Wimmer et al. [WHN18], who prove correctness of a sequential version of the Bellman–Ford algorithm using Isabelle. Their proof strategy is different from ours: they use a framework from Kleinberg and Tardos [KT06] to refine a correct recursive function into an efficient imperative implementation. They first define Bellman–Ford as a recursive function that computes the shortest distances between all vertices using dynamic programming, and then use Isabelle to prove that it returns the shortest path. Then this recursive function is refined into an efficient imperative implementation (see the proof in [VSC]). However, this imperative implementation cannot be naturally parallelized. Moreover, because of the refinement approach, their correctness arguments are different from ours and do not depend on property preservation, which makes them unsuitable for standard deductive code verification.

In the literature there is ample work on the verification of other sequential graph algorithms. Some of these verifications are fully automatic, while others are semi-automatically done by interactive provers. Lammich et al. [LN15, LW19] propose a framework for verifying sequential DFS in Isabelle [Isa]. Chen et al. [CCL+18] provide a formal correctness proof of Tarjan's sequential SCC algorithm using three (both automated and interactive) proof systems: Why3 [Whyb], Coq [Coq] and Isabelle. There is also a collection of verified sequential graph algorithms in Why3 [Whya]. Van de Pol [vdP15] verified the sequential Nested DFS algorithm in Dafny [Daf]. Guéneau et al. [GJCP19] improved Bender et al.'s [BFGT15] incremental cycle detection algorithm to turn it into an online algorithm. They implemented it in OCaml and proved its functional correctness and worst-case amortized asymptotic complexity (using Separation logic combined with Time Credits).

In contrast, there is only limited work on the verification of concurrent graph algorithms. Raad et al. [RHVG16] verified four concurrent graph algorithms using a logic without abstraction (CoLoSL), but their proofs have not been mechanized. Sergey et al. [SNB15] verified a concurrent spanning tree algorithm using Coq.

As far as we are aware, there is no work on mechanized code verification of massively parallel GPU-based graph algorithms. Most similar to our approach is the work by Oortwijn et al. [OHJvdP20, Oor19], who discuss the automated verification of the parallel Nested Depth First Search (NDFS) algorithm of Laarman et al. [LLVDP+11]. Although they are the first to provide a mechanical proof of a parallel graph algorithm, their target is not massively parallel programs on GPUs.

5.7 Conclusion

Graph algorithms play an important role in solving many real-world problems. This chapter shows how to mechanically prove correctness of the parallel Bellman–Ford GPU algorithm, with VerCors. To the best of our knowledge, this is the first work on automatic code verification of this algorithm.

Since we prove the general classic Bellman–Ford algorithm without applying GPU optimization techniques, we plan to investigate how to reuse the current proof for the optimized implementations. Moreover, we also would like to investigate how we can generate part of the annotations automatically.

CHAPTER 6

A Generic Approach to the Verification of the Permutation Property of Sequential and Parallel Swap-based Sorting Algorithms

So far we discussed the verification of some fundamental algorithms on GPUs as case studies. Another fundamental operation in computer science is sorting, for which many sequential and parallel algorithms have been proposed in the literature. Swap-based sorting algorithms are one category of sorting algorithms where elements are swapped repeatedly to achieve the desired order. Since these algorithms are widely used in practice, their (functional) correctness, i.e., proving sortedness and permutation properties, is of utmost importance. However, proving the permutation property using automated program verifiers is much more challenging (than the sortedness property) as the formal definition of this property involves existential quantifiers. In this chapter, we propose a generic pattern to verify the permutation property for any (GPU-based) parallel, *and* also any (CPU-based) sequential swap-based sorting algorithm automatically. To demonstrate our approach, we use VerCors, to verify the permutation property of parallel odd-even transposition sort, and sequential bubble sort, selection sort, insertion sort, quick sort, two in-place merge sorts and TimSort for any arbitrary size of input.

6.1 Introduction

Sorting is one of the fundamental and frequently used operations in computer science. Sorting algorithms take a list of elements as input and rearrange them into a particular order as output. Sorting has many applications in searching, data structures and data bases. Because of its importance, the literature contains many sorting algorithms with different complexity. One category of sorting algorithms is

swap-based sorting, where the elements are swapped repeatedly until the desired order is achieved (e.g., bubble sort).

Because of the increase in the amount of data and emerging multi-core architectures, also parallel versions of sorting algorithms have been proposed. For instance, odd-even transposition sort [Hab72] has been proposed as a parallel version of the bubble sort algorithm. It has been shown that parallel (GPU-based) implementations of sorting algorithms [SHG09, SA08, MG10, KW05, GZ06, GGKM06] outperform their sequential (CPU-based) counterparts.

Due to the frequent use of both sequential and parallel sorting algorithms, their correctness is of utmost importance, which means that they must have the following properties: (1) sortedness: the output is ordered, and (2) permutation: the output is a permutation of the input (i.e., the elements do not change).

Using deductive program verification, proving the permutation property is harder than the sortedness property. This might be surprising at first glance, since in the swap-based sorting algorithms, the main operation is only swapping two elements repeatedly. But the permutation property typically requires reasoning about existential quantifiers, which is challenging for the underlying automated theorem provers.

As discussed by Filliatre [Fil11], there are three common solutions: (1) in a higher order logic, one can state the existence of a bijection; (2) one can use multisets, a collection of unordered lists of elements where multiple instances can occur; and (3) one can define a permutation as the smallest equivalence relation containing the transpositions (i.e., the exchanges of elements). In [Fil11, FM99], it is shown that the third approach is the best solution for automated proofs of the permutation property, but it is still not easy to define it formally. The literature contains various examples of permutation proofs, following the third approach [BSSU17, Sch16a, TM11, Fil13, FM99, TGK09]. In these papers, a permutation is formally defined and some of its properties (e.g., transitivity) are proved using deductive program verifiers, such as KeY [ABB+16] or Why3 [FP13] or interactive theorem provers like Coq [Coq].

In this chapter, we recognize that there is a *uniform pattern* and we exploit this to prove the permutation property of any sequential and parallel swap-based sorting algorithm. Concretely, we use VerCors verifier to prove the permutation property of parallel odd-even transposition sort, sequential bubble sort, selection sort, insertion sort, quick sort, two in-place merge sorts. There are several advantages of our approach w.r.t. the previous work. First, none of the existing papers verified the permutation property of embarrassingly parallel sorting algorithms (e.g., in GPGPU). We demonstrate that our uniform approach also works for such algorithms by proving the permutation property of the parallel odd-even transposition

sort algorithm. Second, the technique works for all languages supported in the tool such as C and Java and OpenCL/CUDA, which means it is possible to prove the permutation property of sorting algorithms in a variety of real-world languages. Third, in our permutation proof pattern, we use ghost variables to keep track of value changes, which can be reused when establishing sortedness. Forth, we illustrate the generality of our approach by proving the permutation property of a vast collection of well-known sorting algorithms all together in one place.

In the rest of the chapter, we first discusse the proposed generic approach to prove the permutation property. Then, we use the pattern to apply it to an extensive collection of well-known sorting algorithms.

6.2 Permutation Verification of Swap-based Sorting

In this section, we describe our generic approach to verify the permutation property of sequential and parallel swap-based sorting algorithms.

6.2.1 Swap-based Sorting Algorithms

An algorithm is a swap-based sorting algorithm if *by only swapping* the elements it satisfies the following:

- INPUT: An array *Input* of integers[1] of size N.
- OUTPUT: An array *Output* of integers of size N such that
  - sortedness: $\forall i.\ 0 \leq i < N-1$: $Output[i] \leq Output[i+1]$
  - permutation: $\forall e \in Input$: $occurrence(Input, e) = occurrence(Output, e)$

where $occurrence(A, e)$ counts the number of occurrences of e in A.

6.2.2 Functional Correctness of Swap-based Sorting Algorithms

To prove the correctness of swap-based sorting algorithms, we use ghost variables, in particular as sequences in VerCors. The most important advantage of using ghost variables is that it allows us to reason about *both* sortedness and permutation properties in the same specification. Moreover, establishing the proof based on ghost sequences helps us to also apply our technique on other data types rather

[1] We specify the type of *Input* as integers, but it can be other types as well.

Algorithm 6.1 Sequential

```
1: inv Output == inp_seq_cur
2: inv Input == inp_seq_chain[0]
3: inv sortedness properties
4: loop(0..M)
       ⋮
5:     swap(Output, i, j)
6:     inp_seq_cur = swap-seq(
7:                   inp_seq_cur, i, j)
       ⋮
8:     inp_seq_chain +=
9:     seq<seq<int>>{inp_seq_cur}
10: end loop
```

Algorithm 6.2 Parallel

```
1: inv Output == inp_seq_cur
2: inv Input == inp_seq_chain[0]
3: inv sortedness properties
4: par(tid = 0..K)
       ⋮
5:   swap(Output, f1(tid), f2(tid))
6:   atomic
7:     inp_seq_cur = swap-seq(
8:     inp_seq_cur, f1(tid), f2(tid))
9:   end atomic
       ⋮
10: end par
11: inp_seq_chain +=
12:   seq<seq<int>>{inp_seq_cur}
```

Figure 6.1: Annotated pseudocode of sequential and parallel swap-based sorting algorithms.

than arrays such as linked lists. To demonstrate that, first we discuss which ghost variables we define and how they are beneficial in verifying sortedness. Then, we explain in detail how these ghost variables can be used to describe a generic verification pattern to verify the permutation property.

Figure 6.1 provides general sketches of the core (annotated) part of sequential and parallel swap-based sorting algorithms. Initially, we assume that *Input* and *Output* contain the same elements. The key operation in both sequential and parallel algorithms is the *swap* function where two elements are swapped. In the sequential algorithms, there is at most one swap at a time, but there might be multiple swaps in one iteration (e.g., sequential odd-even sort). In the parallel version there might be multiple simultaneous swaps, where f_1 and f_2 are two functions that assign a thread to two elements for swapping according to a thread id (i.e., *tid*). Note that in the parallel version, there can be a loop inside or outside the *par* block. By swapping the elements, a new rearrangement of the input array is generated. To keep track of these rearrangements of the elements, we define a ghost variable, *inp_seq_chain* (type sequence of sequences), as a chain of sequences that the first sequence in this chain is *Input*. We also define another ghost variable, *inp_seq_cur*, as the sequence that always stores the current rearrangement (i.e., last sequence in the chain).

Next, in the sequential version, we define a function, *swap-seq* and apply it to *inp_seq_cur* to update the current sequence exactly in the same way as the *swap* function does over *Output* (lines 6-7). Finally, we specify a loop invariant that shows that the array and the sequence are the same in each iteration (line 1). Moreover, in each iteration, after the swapping(s), we add the new current sequence to the chain of sequences (lines 8-9)[2]. This proposed pattern is also applicable for proving the correctness of recursive swap-based sorting algorithms (e.g., quick sort). In the parallel version, the principle is the same, but as there might be simultaneous swaps, we update the current sequence atomically (lines 6-9). Note that the exact location where we add the updated sequence to the chain depends on the algorithms and might be different from the sketches. For instance, if there is a loop inside the parallel block then we add the sequence to the chain at the end of the loop inside the parallel block. Notice that in the parallel algorithm (Figure 6.1) the *swap* function over *Output* (line 5) is outside the atomic block. This matches for instance the parallel odd-even transposition sort. However, in other parallel sorting algorithms, we might need to include the *swap* function inside the atomic block to avoid data races. Note that in the tool we use permission-based separation logic to prove data-race freedom of the algorithms.

These constructs allow us to prove the sortedness and permutation properties of any sequential and parallel swap-based sorting algorithm. By defining a chain of sequences, a user can provide key properties as invariants to reason about how the values change in the chain from *Input* to the last sequence (which is *Output*) to prove sortedness (line 3). To prove the permutation property, first we define a function *occurrence* (as shown in Figure 6.2) that counts the number of occurrences of an element in a sequence. The postcondition of the function specifies (the boundary of) the result of the function in general (line 1) and in three different conditions where the element exists in the sequence (line 5) or does not exist (lines 2) or the sequence only contains that element (lines 3-4). Note that there is a hidden existential quantifier in line 5.

Next, we define a predicate that states that a sequence is a permutation of another sequence if and only if the size of both are the same and the number of occurrences of each element in both are the same (Figure 6.3).

Next, we use VerCors to prove a property that for any sequence if we swap two arbitrary elements, the result is a permutation of the original sequence:

Property 6.1 For any sequence xs:
$(\forall i, j.\, 0 \leq i \leq j < |xs| :$
$(\forall l.\, 0 \leq l < |xs| : \mathit{occurrence}(xs, xs[l]) = \mathit{occurrence}(\mathit{swap\text{-}seq}(xs, i, j), xs[l])))$.

[2]Note that depending on the algorithm, the new arrangement might be added to the chain after one swap *or* multiple swaps.

```
1 /*@ ens \result ≥ 0 && \result ≤ |xs|;
2     ens (\forall int i; 0 ≤ i && i < |xs|; element != xs[i]) <==> \result == 0;
3     ens (\forall int i; 0 ≤ i && i < |xs|; element == xs[i]) <==>
4            \result == |xs|;
5     ens element in xs <==> \result > 0; @*/
6 pure int occurrence(seq<int> xs, int element) = |xs| == 0 ? 0 :
7    ( head(xs) == element ? (1+occurrence(tail(xs), element)) :
8      occurrence(tail(xs), element) );
```

Figure 6.2: The *occurrence* function.

```
1 pure bool permutation(seq<int> xs, seq<int> ys) = (|xs| == |ys|) &&
2  (\forall int i; 0 ≤ i && i < |xs|; occurrence(xs, xs[i]) == occurrence(ys, xs[i]));
```

Figure 6.3: The *permutation* function.

Proof. We define a lemma in VerCors to prove the property. We explain the steps that we have in the lemma (in VerCors) exactly as implemented.

If i equals to j both xs and $swap\text{-}seq(xs, i, j)$ are the same. Thus, the property holds and VerCors can infer it. If i is less than j, we split xs and $swap\text{-}seq(xs, i, j)$ into disjoint sequences in VerCors according to i and j as follows:

$$xs = xs[0..i-1] + xs[i] + xs[i+1..j-1] + xs[j] + xs[j+1..|xs|-1] \tag{6.1.1}$$

$$swap\text{-}seq(xs, i, j) = xs[0..i-1]+xs[j]+xs[i+1..j-1]+xs[i]+xs[j+1..|xs|-1] \tag{6.1.2}$$

We rewrite (6.1.1) and (6.1.2) in terms of the *occurrence* function and prove (by another lemma in VerCors) that this function distributes over concatenation of sequences as follows:

$$\begin{aligned} occurrence(xs + ys + ... + ts, element) = occurrence(xs, element) + \\ occurrence(ys, element) + ... + occurrence(ts, element) \end{aligned} \tag{6.1.3}$$

Then we note that we have the following equalities:

$$\begin{aligned} occurrence(xs, element) = occurrence(xs[0..i-1] + xs[i] + \\ xs[i+1..j-1] + xs[j] + xs[j+1..|xs|-1], element) \end{aligned} \tag{6.1.4}$$

$$\begin{aligned} occurrence(swap\text{-}seq(xs, i, j), element) = occurrence(xs[0..i-1] + \\ xs[j] + xs[i+1..j-1] + xs[i] + xs[j+1..|xs|-1], element) \end{aligned} \tag{6.1.5}$$

By applying property (6.1.3) to equations (6.1.4) and (6.1.5) and using commutativity and associativity of "+" on integers, VerCors can conclude that both right-hand sides of (6.1.4) and (6.1.5) are equal, hence also their left-hand sides are equal. □

This allows us to specify that after each swap, the new sequence is a permutation of the previous one. As a corollary we prove that the *occurrence* function is symmetric:

Corollary 6.1 For any equal-sized sequences xs, ys and ts:
$(\forall i, j.\, 0 \leq i \leq j < |\mathit{swap\text{-}seq}(xs, i, j)| : (\forall l.\, 0 \leq l < |\mathit{swap\text{-}seq}(xs, i, j)| :$
$\mathit{occurrence}(\mathit{swap\text{-}seq}(xs, i, j), \mathit{swap\text{-}seq}(xs, i, j)[l]) =$
$\mathit{occurrence}(xs, \mathit{swap\text{-}seq}(xs, i, j)[l]))).$

Property 6.1 does not specify that the current sequence is a permutation of the input array (i.e., the first sequence in the chain). To establish that, we use VerCors to also prove that the *occurrence* function is transitive:

Property 6.2 For any equal-sized sequences xs, ys and ts:
$(\ (\forall l.\, 0 \leq l < |xs| \rightarrow \mathit{occurrence}(xs, xs[l]) = \mathit{occurrence}(ys, xs[l]))\ \wedge$
$(\forall l.\, 0 \leq l < |ys| \rightarrow \mathit{occurrence}(ys, ys[l]) = \mathit{occurrence}(ts, ys[l]))\ \ ) \Rightarrow$
$(\forall l.\, 0 \leq l < |xs| \rightarrow \mathit{occurrence}(xs, xs[l]) = \mathit{occurrence}(ts, xs[l])).$

Proof. The proof is straightforward. The number of occurrences of each element in xs is the same as in ys and the number of occurrences of elements in ys is the same as in ts. From **Corollary 3.1**, we have the same number of occurrences of each element in ys as in xs and the same number of occurrences of elements in ts as in ys. Since all of them have the same length, VerCors can infer (without intermediate proof steps) that the number of occurrences of each element in xs is the same as in ts. □

Using **Properties 6.1** and **6.2** we can show that the permutation property is preserved for each new rearrangement of the input array during the algorithms:

Permutation Invariant After each swap in the sequential and parallel swap-based sorting algorithms $\mathit{permutation}(\mathit{inp_seq_chain}[0], \mathit{inp_seq_cur})$ holds.

6.3 Case Studies: Proving Permutation of Swap-based Sorting Algorithms

Figure 6.4 illustrates the preservation of the permutation invariant. At the beginning, $\mathit{inp_seq_chain}[0]$ and $\mathit{inp_seq_cur}$ are equal to Input, hence the invariant holds. Assume that the invariant holds between sequences $\mathit{inp_seq_chain}[0]$

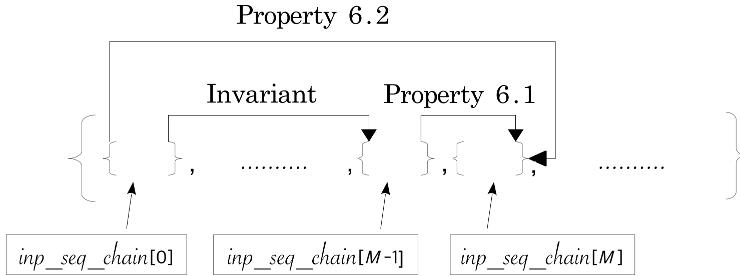

Figure 6.4: An example of preserving permutation property after each sequence.

Algorithm 6.1 Sequential

```
1:  inv Output == inp_seq_cur
2:  inv Input == inp_seq_chain[0]
3:  inv sortedness properties
4:  inv permutation(
5:  inp_seq_chain[0], inp_seq_cur)
6:  loop(0..M)
        ⋮
7:      swap(Output, i, j)
8:      Applying Prop. 6.1 to
9:      inp_seq_cur and Prop. 6.2 to
10:     inp_seq_chain[0], inp_seq_cur
11:     and swap-seq(inp_seq_cur, i, j)
12:     inp_seq_cur = swap-seq(
13:                 inp_seq_cur, i, j)
        ⋮
14:     inp_seq_chain +=
15:     seq<seq<int>>{inp_seq_cur}
16: end loop
```

Algorithm 6.2 Parallel

```
1:  inv Output == inp_seq_cur
2:  inv Input == inp_seq_chain[0]
3:  inv sortedness properties
4:  inv permutation(
5:  inp_seq_chain[0], inp_seq_cur)
6:  par(tid = 0..K)
        ⋮
7:    swap(Output, f1(tid), f2(tid))
8:    atomic
9:       Applying Prop. 6.1 to
10:      inp_seq_cur and Prop. 6.2 to
11:      inp_seq_chain[0], inp_seq_cur
12:      and swap-seq(inp_seq_cur,
13:      f1(tid), f2(tid))
14:      inp_seq_cur = swap-seq(
15:      inp_seq_cur, f1(tid), f2(tid))
16:   end atomic
        ⋮
17: end par
18: inp_seq_chain +=
19:  seq<seq<int>>{inp_seq_cur}
```

Figure 6.5: Annotated pseudocode of sequential and parallel swap-based sorting algorithms.

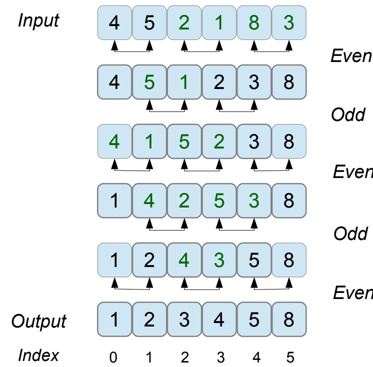

Figure 6.6: An example of odd-even transposition sort. Values in green should be swapped.

(which is *Input*) and *inp_seq_chain*[M-1] (which is *inp_seq_cur*). Then, after each swap, we can apply **Properties 6.1** and **6.2** to show that the invariant is preserved. Therefore, after the last swap when we add the updated *inp_seq_cur* (*inp_seq_chain*[M]) to the chain, the invariant still holds. In the pseudocode of sequential and parallel swap-based sorting algorithms, we only need to apply the two properties before a swap. This is sufficient to prove the permutation property of the algorithms. Figure 6.5 illustrates this and completes the generic pattern. As we can see, the pattern can be generated automatically and the only parts that a user needs to fill out are the arguments i and j which are specific to the algorithms.

In this section, we show how we apply the technique described above, to verify multiple parallel and sequential swap-based sorting algorithms. In particular, we discuss how we prove the permutation property of parallel odd even transposition sort as well as sequential bubble sort, selection sort and insertion sort. Moreover, we illustrate how we use our approach to prove the permutation property of recursive in-place sorting algorithms quick sort and merge sort. In addition, we benefit from the verification of insertion sort and merge sort to verify the permutation property of TimSort.

6.3.1 Permutation Verification of Parallel Odd-Even Transposition Sort

Odd-even transposition sort is a parallel version of bubble sort. It consists of two phases: odd and even. Algorithm 6.3 presents the annotated pseudocode while Figure 6.6 shows an example of the execution of the algorithm. In the algorithm,

Algorithm 6.3 Parallel odd-even transposition sort

```
1:  bool isSorted = false
2:  inv Output == inp_seq_cur
3:  inv Input == inp_seq_chain[0]
4:  inv permutation(inp_seq_chain[0], inp_seq_cur)
5:  while !isSorted do
6:    isSorted = true
7:    Par(tid = 0..N/2) // Thread id from 0 to (N/2)-1
8:      if 2×tid+1 < N && Output[2×tid] > Output[2×tid+1] then
9:        Swap(Output, 2×tid, 2×tid+1)
10:       atomic
11:         Applying Prop. 6.1 to inp_seq_cur and Prop. 6.2 to inp_seq_chain[0],
12:         inp_seq_cur and swap-seq(inp_seq_cur, 2×tid, 2×tid+1)
13:         inp_seq_cur = swap-seq(inp_seq_cur, 2×tid, 2×tid+1)
14:       end atomic
15:       isSorted = false
16:   end par
17:   inp_seq_chain = inp_seq_chain + seq<seq<int>>{inp_seq_cur}
18:   Par(tid = 0..N/2) // Thread id from 0 to (N/2)-1
19:     if 2×tid+2 < N && Output[2×tid+1] > Output[2×tid+2] then
20:       Swap(Output, 2×tid+1, 2×tid+2)
21:       atomic
22:         Applying Prop. 6.1 to inp_seq_cur and Prop. 6.2 to inp_seq_chain[0],
23:         inp_seq_cur and swap-seq(inp_seq_cur, 2×tid+1, 2×tid+2)
24:         inp_seq_cur = swap-seq(inp_seq_cur, 2×tid+1, 2×tid+2)
25:       end atomic
26:       isSorted = false
27:   end par
28:   inp_seq_chain = inp_seq_chain + seq<seq<int>>{inp_seq_cur}
```

Output is initialized to *Input*. In the even phase (lines 7-18), even locations ($2 \times tid$) are compared to their right neighbor ($2 \times tid+1$) in line 8 and swapped if they are greater (line 9). In the odd phase (lines 20-31), odd locations ($2 \times tid+1$) are compared and swapped in the same way with their right neighbor ($2 \times tid+2$) in lines 21-22. This process repeats inside a loop (line 5) until all elements are sorted, i.e., there is no swap (indicated by a boolean, *isSorted*).

To prove permutation, we use the pattern proposed in Algorithm 2 (Figure 6.5) for both phases. We only fill out the two locations that need to be swapped (i.e., $2 \times tid$ and $2 \times tid+1$ in line 14, $2 \times tid+1$ and $2 \times tid+2$ in line 27). By applying the properties before a swap (lines 12-14 and 25-27), we establish the permutation

Algorithm 6.4 Bubble sort

```
1: inv Output == inp_seq_cur
2: inv Input == inp_seq_chain[0]
3: inv permutation(
4: inp_seq_chain[0], inp_seq_cur)
5: loop(k: 0..N-2)
6:   inv Output == inp_seq_cur
7:   inv Input == inp_seq_chain[0]
8:   inv permutation(
9:   inp_seq_chain[0], inp_seq_cur)
10:  loop(t: 0..N-k-2)
11:    if Output[t] > Output[t+1]
12:      swap(Output, t, t+1)
13:      Applying Prop. 6.1 to
14:      inp_seq_cur and Prop. 6.2 to
15:      inp_seq_chain[0], inp_seq_cur,
16:      swap-seq(inp_seq_cur, t, t+1)
17:      inp_seq_cur = swap-seq(
18:                inp_seq_cur, t, t+1)
19:    end if
20:  end loop
21:  inp_seq_chain +=
22:   seq<seq<int>>{inp_seq_cur}
23: end loop
```

Algorithm 6.5 Selection sort

```
1: inv Output == inp_seq_cur
2: inv Input == inp_seq_chain[0]
3: inv permutation(
4: inp_seq_chain[0], inp_seq_cur)
5: loop(k: 0..N-2)
6:   minIdx = k
7:   inv Output == inp_seq_cur
8:   inv Input == inp_seq_chain[0]
9:   inv permutation(
10:  inp_seq_chain[0], inp_seq_cur)
11:  loop(t: k+1..N-1)
12:    if Output[t] < Output[minIdx]
13:      minIdx = t
14:  end loop
15:  swap(Output, k, minIdx)
16:  Applying Prop. 6.1 to
17:  inp_seq_cur and Prop. 6.2 to
18:  inp_seq_chain[0], inp_seq_cur,
19:  swap-seq(inp_seq_cur, k, minIdx)
20:  inp_seq_cur = swap-seq(
21:          inp_seq_cur, k, minIdx)
22:  inp_seq_chain +=
23:   seq<seq<int>>{inp_seq_cur}
24: end loop
```

Figure 6.7: Proving permutation property of bubble and selection sort using the proposed pattern.

property indicated as an invariant in line 4.

6.3.2 Permutation Verification of Bubble, Selection and Insertion Sort

This section uses our generic pattern to verify the permutation property of bubble sort, selection sort, and insertion sort, as illustrated in Figure 6.7 and Algorithm 6.6. In bubble and insertion sort, there are two nested loops and a swap happens inside the inner loop. In selection sort, there are also two nested loops, but a swap happens in the outer loop. In all three algorithms, we follow exactly the

Algorithm 6.6 Insertion sort

```
1:  inv Output == inp_seq_cur
2:  inv Input == inp_seq_chain[0]
3:  inv permutation(inp_seq_chain[0], inp_seq_cur)
4:  loop(k: 1..N-1)
5:    inv Output == inp_seq_cur
6:    inv Input == inp_seq_chain[0]
7:    inv permutation(inp_seq_chain[0], inp_seq_cur)
8:    loop(t: k..1)
9:      if Output[t-1] > Output[t]
10:       swap(Output, t-1, t)
11:       Applying Prop. 6.1 to inp_seq_cur and Prop. 6.2 to
12:       inp_seq_chain[0], inp_seq_cur and swap-seq(inp_seq_cur, t-1, t)
13:       inp_seq_cur = swap-seq(inp_seq_cur, t-1, t)
14:     end if
15:   end loop
16:   inp_seq_chain = inp_seq_chain + seq<seq<int>>{inp_seq_cur}
17: end loop
```

approach as discussed in Section 6.2. We have the same invariants (for both loops) and we apply the same properties before a swap. The only differences are the two locations of elements to be swapped, which we set according to the algorithms themselves.

6.3.3 Permutation Verification of Quick Sort

The proposed pattern can also be used for recursive in-place sorting algorithms. To show this, we use the pattern to verify the permutation property of the quick sort algorithm as illustrated in Algorithm 6.7. This recursive algorithm is initialized with $low = 0$ and $high = |Output|$-1. Each recursive call puts the last element (the pivot), in the correct position in the sorted array in such a way that all smaller elements will be to the left of the pivot and all larger elements will be to the right of the pivot. The function recursively applies the same function to both subarrays to the left and right of the pivot (lines 23-24), resulting in a sorted array. As we can see there are two swaps, one inside the loop and the other one outside the loop (lines 9 and 16). Again, we apply the properties before each swap and we add the new sequence (i.e., *inp_seq_cur*) to the chain (i.e., *inp_seq_chain*) after the second swap (line 21). In this way, we prove permutation of the quick sort algorithm.

Algorithm 6.7 Quick sort

```
1: if low < high
2:   pivot = Output[high], idx = low-1
3: inv Output == inp_seq_cur
4: inv Input == inp_seq_chain[0]
5: inv permutation(inp_seq_chain[0], inp_seq_cur)
6:   loop(k: low..high-1)
7:     if Output[k] ≤ pivot
8:       idx++
9:       swap(Output, idx, k)
10:      Applying Prop. 6.1 to inp_seq_cur and Prop. 6.2 to
11:      inp_seq_chain[0], inp_seq_cur and swap-seq(inp_seq_cur, idx, k)
12:      inp_seq_cur = swap-seq(inp_seq_cur, idx, k)
13:    end if
14:  end loop
15:  swap(Output, idx+1, high)
16:  Applying Prop. 6.1 to inp_seq_cur and Prop. 6.2 to
17:  inp_seq_chain[0], inp_seq_cur and swap-seq(inp_seq_cur, idx+1, high)
18:  inp_seq_cur = swap-seq(inp_seq_cur, idx+1, high)
19:  inp_seq_chain = inp_seq_chain + seq<seq<int>>{inp_seq_cur}
20:  pivotIdx = idx+1
21:  Recursive call: Quick sort(Output, low, pivotIdx-1)
22:  Recursive call: Quick sort(Output, pivotIdx+1, high)
23: end if
```

6.3.4 Permutation Verification of Merge Sort

Merge sort is another example of a recursive sort that splits the elements into smaller parts (recursively) and merges them into a sorted array. Thus, the main part of the algorithm is merging two sorted subarrays. Figure 6.8 presents the (annotated) pseudocode and an example of an in-place merging. The example shows how the merge operates on two sorted subarrays (from indices 0-3 and 4-7). Initially, *start* points to the first element in the array (i.e., index 0), and *right* indicates the last element (i.e., index N-1). Moreover, the variable $start2$ points to the first location in the right subarray (i.e., index $mid+1$) where *mid* equals $(start+right)/2$. Then, the elements in these two locations are compared and if $Output[start] > Output[start2]$, we should insert the element in location $start2$ into location *start* by shifting all elements in between by one location to the right (lines 12-25 of the pseudocode). Otherwise, we increase *start* by 1 (line 11). This process repeats until the array is sorted. As we can see in the pseudocode, the shifting is implemented as swapping two adjacent elements from location $start2$

Algorithm 6.8 In-place merge1

```
 1: start2 = mid+1
 2: if Output[mid] ≤ Output[start2]
 3:   return
 4: end if
 5: inv Output == inp_seq_cur
 6: inv Input == inp_seq_chain[0]
 7: inv permutation(
 8: inp_seq_chain[0], inp_seq_cur)
 9: while(start ≤ mid && start2 ≤ right)
10:   if Output[start] ≤ Output[start2]
11:     start+=1
12:   else
13:     idx = start2
14:     while(idx != start)
15:       swap(Output, idx-1, idx)
16:       Applying Prop. 6.1 to
17:       inp_seq_cur and Prop. 6.2 to
18:       inp_seq_chain[0], inp_seq_cur,
19:       swap-seq(inp_seq_cur, idx-1, idx)
20:       inp_seq_cur = swap-seq(
21:         inp_seq_cur, idx-1, idx)
22:       idx-=1
23:     end while
24:     start+=1, mid+=1, start2+=1
25:   end if else
26: end while
27: inp_seq_chain +=
28:   seq<seq<int>>{inp_seq_cur}
```

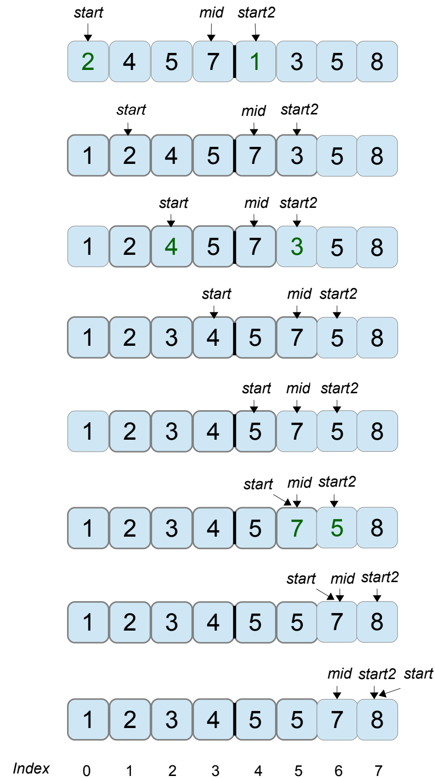

Figure 6.8: Annotated pseudocode of an in-place merging (left) and an example (right). Green values in the example indicates that the comparison implies some swaps.

to *start* consecutively. In this way, we can reuse the proposed pattern again to verify the permutation property of this algorithm.

Figure 6.9 presents another (annotated) pseudocode and an example of an in-place merge [DĎ88, KK04] which is more efficient than the previous one in complexity. In this algorithm, initially *left* points to index zero, *right* equals N and the two

Algorithm 6.9 In-place merge2

1: **if** $left{=}{=}mid \mid\mid mid{=}{=}right$
2: **return**
3: **end if**
4: $start = mid\text{-}1,\ end = mid$
5: **while**($left \leq start$ && $end < right$ &&
6: $Output[start] > Output[end]$)
7: $start{-}{=}1,\ end{+}{=}1$
8: **end while**
9: inv $Output == inp_seq_cur$
10: inv $Input == inp_seq_chain[0]$
11: inv $permutation($
12: $inp_seq_chain[0],\ inp_seq_cur)$
13: **loop**(k: 0..end-mid-1)
14: $swap(Output,\ 2{\times}mid\text{-}end{+}k,\ mid{+}k)$
15: Applying **Prop. 6.1** to inp_seq_cur
16: and **Prop. 6.2** to $inp_seq_chain[0]$,
17: inp_seq_cur and $swap\text{-}seq($
18: $inp_seq_cur,\ 2{\times}mid\text{-}end{+}k,\ mid{+}k)$
19: $inp_seq_cur = swap\text{-}seq($
20: $inp_seq_cur,\ 2{\times}mid\text{-}end{+}k,\ mid{+}k)$
21: **end loop**
22: Recursive call:
23: $merge2(Output,\ left,\ start{+}1,\ mid)$
24: Recursive call:
25: $merge2(Output,\ mid,\ end,\ right)$

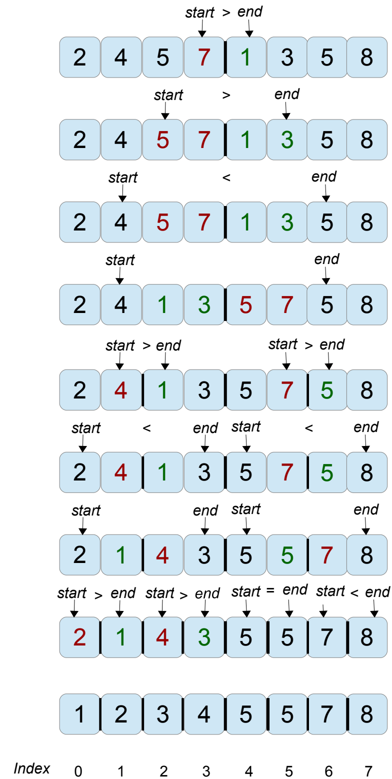

Figure 6.9: Another in-place merging: annotated pseudocode (left) and an example (right). Red values should be swapped with green values in each recursion.

variables, *start* and *end* point to the last and first elements of the subarrays, respectively[3]. Variable *start* decreases and *end* increases by one until $Output[start] \leq Output[end]$ (lines 5-8). When $Output[start] \leq Output[end]$, we should swap the two subarrays in ranges $(start,\ mid)$ (as red values in the example on the right) and $[mid,\ end)$ (as green values in the example), as in line 13-21 of the pseudocode. As we can see, we do swaps one by one between the first elements in the two ranges, then between the second elements, and so on. As a result, all elements in the left subarray become smaller than all the elements in the right sub-

[3] In the example, the middle element initially is in index 4, because *right* equals N.

array. This means that the subarrays are now independent and the same process can be applied for both of them. Therefore, we recursively call this process for the two subarrays to sort the full array (lines 22-25). Thus, the merge function is also recursive in addition to the main function of merge sort. Since there are swaps in this algorithm, we reuse the generic pattern and verify the permutation property of this algorithm as well. The only point is that, since the merging function is recursive, we add the new rearrangement of the elements (i.e., *inp_seq_cur*) to the chain (i.e., *inp_seq_chain*) in the main function of merge sort instead of in the merge function itself.

6.3.5 Permutation Verification of TimSort

Amongst the sorting algorithms, insertion sort performs better in practice when the number of elements are small (e.g., 64). Merge sort performs well when the size of two subarrays is power of 2. TimSort [Pet02] benefits from this as a combination of insertion and merge sort. Algorithm 6.10 presents a simplified (annotated) version of this algorithm. It first sorts small groups of elements (e.g., 64) as runs (lines 5-9). Then, the algorithm repeatedly merges these equal-size sorted runs using the merging function (lines 14-29). That means, in the first iteration, the algorithm merges each two consecutive runs into a larger size of runs, and it repeats merging for each two consecutive (larger) runs until the full array is sorted. Note that we can use both merge functions discussed above for TimSort.

Since we already proved permutation of insertion and merge sort, we can easily prove the permutation property of TimSort. In fact, we prove permutation of two (in-place) TimSort algorithms using the two verified merge functions.

6.4 Related Work

There are several papers on proving sortedness and permutation properties of concrete sorting algorithms. In [Sch16a, TM11, Fil13], the authors prove correctness of various sorting algorithms using the Why3 [FP13] platform[4]. Why3 is a program verifier which has its own language for programming and specification (i.e., WhyML) based on first-order logic. It is mainly used as backend for other verifiers. To prove sortedness and permutation properties, suitable lemmas and invariants are defined and used in the extensive Why3 library[5]. However, they do not propose a generic approach to verify sortedness and permutation properties of parallel sorting algorithms. They only prove the correctness of several sequential sorting

[4] The verified sorting algorithms using Why3 are available at `http://pauillac.inria.fr/~levy/why3/sorting`.

[5] See `http://why3.lri.fr/stdlib/array.html`.

Algorithm 6.10 In-place TimSort

```
1: RUN = 64, k = 0
2: inv Output == inp_seq_cur
3: inv Input == inp_seq_chain[0]
4: inv permutation(inp_seq_chain[0], inp_seq_cur)
5: while(k < N)
6:     end = min(k+RUN-1, N-1)
7:     Insertion sort(Output[k...end]
8:     k+=RUN
9: end while
10: chunk = RUN
11: inv Output == inp_seq_cur
12: inv Input == inp_seq_chain[0]
13: inv permutation(inp_seq_chain[0], inp_seq_cur)
14: while(chunk < N)
15:     left = 0
16: inv Output == inp_seq_cur
17: inv Input == inp_seq_chain[0]
18: inv permutation(inp_seq_chain[0], inp_seq_cur)
19:     while(left < N)
20:         mid = left+chunk-1, right = min(left+2×chunk-1, N-1)
21:         if(mid < N-1)
22:             merge1(Output, left, mid, right) // Or merge2(Output, left, mid+1, right+1)
23:         end if
24:         inp_seq_chain = inp_seq_chain + seq<seq<int>>{inp_seq_cur}
25:         left+=(2×chunk)
26:     end while
27:     chunk×=2
28: end while
```

algorithms.

Beckert et al. [BSSU17] prove JDK's dual pivot quick sort algorithm using KeY verifier [ABB$^+$16]. KeY is a program verifier for Java programs. The annotated Java programs are transformed into the internal dynamic logic representation of KeY, and then proof obligations are discharged to its first-order theorem prover which is based on sequent calculus. They benefit from sequences to prove sortedness and permutation properties of the algorithm. To prove the permutation property, they provide suitable invariants and prove some lemmas in the tool. They mention that proving the permutation property is by far the hardest part of their verification, which requires more interaction with the tool than the sort-

edness property. They neither outline a generic pattern nor verify any parallel sorting algorithms. In addition, they only verify quick sort in Java.

De Gouw et al. [dGRdB+15] found a bug in the TimSort implementation in one of OpenJDK's libraries while verifying the code using KeY. They show the effectiveness of (semi) automatic verification in finding bugs in a complex algorithm.

Filliâtre et al. [FM99] verify three sorting algorithms, insertion sort, quick sort and heap sort, in the Coq proof assistant. To prove the permutation property, they propose to express that the set of permutations is the smallest equivalence relation containing the transpositions (i.e., the exchanges of elements).

Tushkanova et al. [TGK09] discuss two specification languages, Java Modeling Language (JML) and Krakatoa Modeling Language (KML), to verify selection sort in Java automatically. To prove the permutation property, they use bags to show that the input and output array have the same content. Their approach is different from ours as we opt to not use bags explicitly to have a uniform pattern for verifying both sortedness and permutation.

6.5 Conclusion

Sorting algorithms are widely used in practice and their correctness is an important issue. To prove correctness of sorting algorithms, we should prove sortedness and permutation properties. Proving the permutation property is harder than sortedness, because it requires reasoning about existential quantifiers. In this chapter, we propose a uniform approach to verify the permutation property of any sequential and parallel swap-based sorting algorithms. To demonstrate that our technique is generic, we prove the permutation property of parallel odd-even transposition sort, and sequential bubble sort, selection sort, insertion sort, in-place (recursive) quick sort, two merge sorts and TimSort using the VerCorse verifier. As a consequence, the proposed generic pattern in this chapter can be used by other people to verify the permutation property of other sorting algorithms.

As future work, we plan to augment the proofs by providing the sortedness property for complex sorting algorithms such as odd-even transposition sort and TimSort using the proposed pattern.

Part II

Correct GPU Optimizations; annotation transformations

CHAPTER **7**

Annotation-Aware GPU Optmiziations

In the first part of the thesis, we discussed the verification of fundamental algorithms (on GPUs). We proved correctness of unoptimized versions of those algorithms. However, to obtain the best performance out of GPUs, a typical development process involves the manual or semi-automatic application of optimizations prior to compiling the code. That means, programers apply incremental optimizations manually, tailored to the GPU architecture. Applying these optimizations might introduce non-trivial errors to the program. There has been some work to automate the optimization process, but there has been little effort to guarantee that correctness is preserved during optimization. Moreover, in addition to the manual optimizations, the annotations should also be (manually) changed accordingly, which in itself is time-consuming and error-prone. Therefore, the second part of this thesis introduces an *annotation-aware transformation* technique: during program transformation, not only the code should change automatically, but also the corresponding annotations while preserving their provability using deductive verification. Concretely, this chapter demonstrates this approach by applying it to six well-known GPU program optimizations, namely loop unrolling, iteration merging, matrix linearization, data prefetching, tiling and kernel fusion. Moreover, this chapter evaluates this approach, in combination with the VerCors program verifier, to automatically optimize a collection of verified GPU programs, including the case studies in the first part of the thesis, and reverify them.

7.1 Introduction

To reason about GPU programs using deductive technique a program requires to be *manually* augmented with pre- and postcondition-style annotations. However, annotating a program is often time consuming. The more complex a program is, the more challenging it becomes to annotate it. In particular, as a GPU program

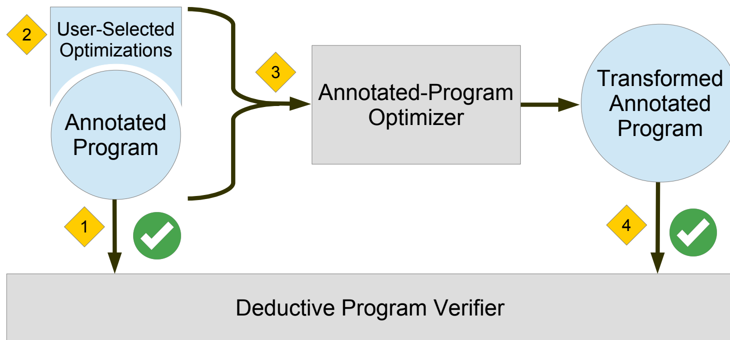

Figure 7.1: Annotation-Aware Program Transformation.

is being optimized repeatedly, its annotations tend to change frequently. This happens because developers should apply incremental optimizations, tailored to the GPU architecture, to achieve the most performance out of GPUs. Unfortunately, this is to a large extent a *manual* activity. The fact that for different GPU devices, the same code tends to require a different sequence of transformations [GGXS+12] makes this procedure even more time consuming and error-prone. Recently, automating this has received some attention, for instance by applying machine learning [AKC+18]. However each optimization might introduce non-trivial errors to the program and little effort has been made to guarantee that functional correctness of programs is preserved by such transformations. Traditionally, this is done by proving the correctness of transformations, i.e., preserving semantic equivalence between the original and the transformed program [Ler06, Ler09]. However, this requires enormous manual effort to formally define and prove each optimization.

This chapter introduces an orthogonal approach that is more feasible and easier to use: we define annotation-aware (source-to-source) transformations such that if an original program is verified, then the program *along with its annotations* are transformed and the resulting annotated optimized program can be verified again. Figure 7.1 illustrates our approach. A program verifier can verify a given annotated program (step 1). The user specifies optimizations to be applied (step 2). This verified program along with the (user-selected) optimizations are given to an annotated-program optimizer (step 3). The result is an optimized program with updated annotations. Finally, the optimized program can still be verified by the same verifier (step 4). In this way, we first automate the optimization of GPU code, to the extent that the developer needs to indicate which optimization needs to be applied where. Second, we apply a code transformation to also transform the related annotations, which means that once the developer has annotated the unoptimized, simpler code, any further optimized version of that code is automatically annotated with updated pre- and postconditions, making it reverifiable. This

avoids having to re-annotate the program every time it is optimized for a specific GPU device.

Interestingly, in some cases the presence of annotations can be exploited to determine whether an optimization is actually applicable, where a compiler cannot determine this. Notice that with this approach, extending it with a new optimization is easier than developing a proof of correctness of the transformation, because each optimization only requires a transformation procedure as we reverify a program each time the optimization is applied using the extant deductive verifiers.

In the rest of chapter, we explain our annotation-aware transformation technique for six GPU optimizations, namely loop unrolling, iteration merging, matrix linearization, data prefetching, tiling and kernel fusion. First, we give a motivational example that shows how to optimize a verified GPU program while preserving its provability. Then, we discuss the implementation of this approach, i.e., our annotated-program optimizer as part of the VerCors verifier for all optimizations. Next, we illustrate this in practice by applying the six optimizations to a collection of verified examples by VerCors, including the complex verified case studies in the first part of the thesis.

7.2 Motivational Example

This section illustrates how to optimize a verified GPU program while preserving its provability. Figure 7.2 shows a GPU program with annotations that is verified by VerCors. The example is written in a simplified version of VerCors' own language PVL. The program initializes an array a, and subsequently updates the values in a, N times. In this program, there are two kernels, both executed by one thread block of a.`length` threads (line 11 and line 17)[1]. In the first kernel (lines 11-12), each thread initializes $a[tid]$ to 0. In the second kernel (lines 17-26), each thread updates $a[tid+1]$ (modulo a.`length`) N times, by adding tid to it. In the main *Host* function, *Kernel1* is called, followed by *Kernel2*.

The kernels, the for-loop and the host function are annotated for verification (in blue). In the example, write permissions are required for all locations in a (line 2). The pre- and postconditions of the first kernel specify that each thread needs write permission for $a[tid]$ (line 9). The postcondition states that $a[tid]$ is set to 0 (line 10). In the second kernel, all threads have write permission for $a[tid+1]$, except thread a.`length`-1 which has write permission for $a[0]$ (line 14). Moreover, it is required that $a[tid+1]$ (modulo a.`length`) is 0 (line 15). For the for-loop

[1] In practice, the size of a block cannot exceed a specific upper-bound, but for this example, we assume that a.`length` is sufficiently small.

```
 1 /*@ context_everywhere N > 0 && N < a.length;
 2     req (\forall* int i; 0 ≤ i && i < a.length; Perm(a[i], 1));
 3     ens (\forall* int i; 0 ≤ i && i < a.length;
 4                                   i != a.length-1 ==> Perm(a[i+1], 1));
 5     ens (\forall* int i; 0 ≤ i && i < a.length; i == a.length-1 ==> Perm(a[0], 1));
 6     ens (\forall int i; 0 ≤ i && i < a.length-1; a[i+1] == N×i);
 7     ens a[0] == N×(a.length-1); @*/
 8 void Host(int[] a, int size, int N) {
 9   /*@ context Perm(a[tid], 1);
10       ens a[tid] == 0; @*/
11   par Kernel1(int tid = 0..a.length)
12   { a[tid] = 0; }
13
14   /*@ context tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
15       req tid != a.length-1 ? a[tid+1] == 0 : a[0] == 0;
16       ens tid != a.length-1 ? a[tid+1] == N×tid : a[0] == N×tid; @*/
17   par Kernel2(int tid = 0..a.length)
18   {
19     /*@ inv k ≥ 0 && k ≤ N;
20         inv tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
21         inv tid != a.length-1 ? a[tid+1] == k×tid : a[0] == k×tid; @*/
22     for(int k = 0; k < N; k++) {
23       if (tid != a.length-1) { a[tid+1] = a[tid+1] + tid; }
24       else { a[0] = a[0] + tid; }
25     }
26   }
27 }
```

Figure 7.2: A verified GPU-style program

(lines 22-25), loop invariants are specified: k is in the range $[0,N]$ (line 19), each thread has write permission for $a[tid+1]$ (modulo a.length) (line 20) and this location always has the value $k \times tid$ (line 21). The postconditions of the second kernel and the host function are similar to this latter invariant.

Figure 7.3 shows an optimized version of the program, with updated annotations to make it verifiable. We have applied three optimizations:

1. *Fusing the two kernels*: in GPU programs, the only *global* synchronisation points (used, for instance, to avoid data races) exist implicitly between kernel launches. However, if such a global synchronisation point is not really needed between two specific kernels, then fusing them gives several benefits, in particular the ability to store intermediate results in (fast) thread-local register memory as opposed to (slow) GPU global memory, and it has a positive effect on power consumption [WLY10]. In the example, the kernels are

```
1  /*@ context_everywhere N > 0 && N < a.length;
2      req (\forall* int i; 0 ≤ i && i < a.length; Perm(a[i], 1));
3      ens (\forall* int i; 0 ≤ i && i < a.length;
4                                          i != a.length-1 ==> Perm(a[i+1], 1));
5      ens (\forall* int i; 0 ≤ i && i < a.length; i == a.length-1 ==> Perm(a[0], 1));
6      ens (\forall int i; 0 ≤ i && i < a.length-1; a[i+1] == N×i);
7      ens a[0] == N×(a.length-1); @*/
8  void Host(int[] a,int size,int N){
9    /*@ req Perm(a[tid], 1);
10       ens tid != a.length-1 ?  Perm(a[tid+1], 1) :  Perm(a[0], 1);
11       ens tid != a.length-1 ?  a[tid+1] == N×tid :  a[0] == N×tid; @*/
12   par Fused_Kernel(int tid = 0..a.length)
13   {
14     a[tid] = 0;
15
16     /*@ req Perm(a[tid], 1);
17         req a[tid] == 0;
18         ens tid != a.length-1 ?  Perm(a[tid+1], 1) :  Perm(a[0], 1);
19         ens tid != a.length-1 ?  a[tid+1] == 0 :  a[0] == 0; @*/
20     barrier(Fused_Kernel)
21
22     int a_reg_0, a_reg_1;
23     if (tid != a.length-1) { a_reg_1 = a[tid+1] } else { a_reg_0 = a[0] }
24     int k = 0;
25     if (tid != a.length-1) { a_reg_1 = a_reg_1 + tid; }
26     else { a_reg_0 = a_reg_0 + tid; }
27     k++;
28
29     /*@ inv k ≥ 0 + 1 && k ≤ N;
30         inv tid != a.length-1 ? Perm(a[tid+1], 1) : Perm(a[0], 1);
31         inv tid != a.length-1 ?  a_reg_1 == k×tid :  a_reg_0 == k×tid; @*/
32     for(k; k < N; k++) {
33       if (tid != a.length-1) { a_reg_1 = a_reg_1 + tid; }
34       else { a_reg_0 = a_reg_0 + tid; }
35     }
36
37     if (tid != a.length-1) { a[tid+1] = a_reg_1 } else { a[0] = a_reg_0 };
38   }
39 }
```

Figure 7.3: An optimized GPU-style program, annotated for verification

combined into *Fused_Kernel*, and a *thread block-local* barrier is introduced (line 20) to avoid data races within the single thread block executing the code.

2. *Using register memory*; register variables can be used to reduce the number of global memory accesses. Here, the use of *a_reg_0* and *a_reg_1* has

been enabled by kernel fusion.

3. *Unrolling the for-loop*; the for-loop has been unrolled once here (lines 22-27). Since GPU threads are very light-weight, compared to CPU threads, any checking of conditions that can be avoided benefits performance. Note that contemporary GPUs do not have branch predictors as in CPUs. When unrolling a loop, this means that fewer checks of the loop-condition are needed. Note that here, we benefit from the knowledge that $N > 0$ (line 1), so we know that the for-loop can be unrolled at least once.

To preserve provability of the optimized program, we changed the annotations, in particular the pre- and postcondition of the fused kernel and the loop invariants (highlighted in Figure 7.3). Moreover, we introduced an annotated barrier (lines 16-20). This example illustrates that reverifying an optimized GPU program requires manual effort, where we typically verify an unoptimized version first. Then, we apply a chain of optimizations step-by-step to obtain more performance. As we can see, applying such optimizations manually in such a way that the optimized program will still be verifiable is non-trivial and tedious. The goal of this chapter is to *automate* this process using tool support.

7.3 GPU Optimizations

This section illustrates our approach on six frequently used GPU optimizations, namely loop unrolling, iteration merging, matrix linearization, data prefetching, tiling and kernel fusion. For each of these optimizations, we first illustrate them by an example. Then, we define a procedure to make the transformation annotation-aware. To focus on the main ideas, the examples in this section are simplified such that some annotations that are not relevant to the transformations, are omitted.

7.3.1 Loop Unrolling

Loop unrolling is a frequently-used optimization technique that is applicable to both GPU and CPU programs. It unrolls some iterations of a loop of a GPU (or CPU) program. This increases the code size, but can have a positive impact on program performance; e.g. see [GGXS+12, MRBS10, Wer19, RKBA+13, vWMBS14] for its impact, specifically on GPU programs. Figure 7.4 shows an example of unrolling an (annotated) loop twice: the body of the loop is duplicated twice before the loop in the optimized program. This has the following effect on the annotations: the loop invariant bounding the loop variable (line 5) changes in the optimized program (line 14). Note that the other loop invariants (i.e., `Inv`(i)) remain the same. Moreover, after each unrolling part, we add all invariants as assertions (lines 8-10) except after the last unroll. This captures that the code

```
1  /*@ context_everywhere N > 1; @*/
2  void Host(int[] arr, int size, int
       N){
3    par Kernel(tid = 0..size){
4      int i = 0;
5      /*@ inv i ≥ 0 && i ≤ N;
6          inv N > 1;
7          inv Inv(i); @*/
8      loop(i < N){
9        int newInt = i;
10       arr[tid] = arr[tid] + newInt;
11       i = i + 1;
12     }
13   }
14 }
```

⇒

```
1  /*@ context_everywhere N > 1; @*/
2  void Host(int[] arr, int size, int
       N){
3    par Kernel(tid = 0..size){
4      int i = 0;
5      int newInt = i;
6      arr[tid] = arr[tid] + newInt;
7      i = i + 1;
8      //@ assert i ≥ 1 && i ≤ N;
9      //@ assert N > 1;
10     //@ assert Inv(i);
11     newInt = i;
12     arr[tid] = arr[tid] + newInt;
13     i = i + 1;
14     /*@ inv i ≥ 2 && i ≤ N;
15         inv N > 1;
16         inv Inv(i); @*/
17     loop(i < N){
18       newInt = i;
19       arr[tid] = arr[tid] + newInt;
20       i = i + 1;
21     }
22   }
23 }
```

Figure 7.4: An example of unrolling a loop 2 times.

produced by unrolling the loop should still satisfy the original loop invariants.

Our approach to loop unrolling is more general than optimization techniques during compilation. For instance, the `unroll` pragma in CUDA [The21a] and the unroll function in Halide [The21b] unroll loops by calculating the number of iterations to see if unrolling is possible, i.e., it should be computable at compile time. This difference is illustrated in Figure 7.4 where N (i.e., the number of iterations) is unknown at compile time. Their approach *cannot automatically* handle this case, while our approach *can automatically* unroll the loop, since annotations (line 1 and line 6) specify the lower-bound of N (provided by the programmer, who knows that this is a valid lower-bound). VerCors verifies that the unrolling is valid.

Figure 7.5 shows an annotated loop template for a verified GPU program. Each thread first initializes the loop variable (line 3). Then it checks the condition (line 9), and according to that, it executes the loop body (line 10) and updates the loop variable (line 11). The annotation indicates the lower and upperbound of the loop variable (line 7) and other properties as invariants (line 8). We would

```
1  void Host(int[] array, int size){
2    par Kernel(tid = 0..size){
3      int i = init(); // The loop variable
4      ⋮
5      //@ assert (i == a) || (i == b); // Depending on initialization of i only one
6                                       // of the conditions is specified
7      /*@ inv i ≥ a && i ≤ b; // The lowerbound of i (a), The upperbound of i (b)
8         inv Inv(i); @*/ // Additional loop invariants
9      loop(cond(i)) { // The loop condition
10       body(i); // The loop body, a sequence of statements in the i^th iteration
11       i = upd(i); // The update function of i, restricted to (i + c), (i - c),
12     }           // (i × c) or (i / c) where c is a positive integer constant²
13   }
14 }
```

Figure 7.5: A general template of a loop inside a kernel.

like to automatically unroll the loop k times and preserve the provability of the program. To accomplish this, we follow the procedure in Figure 7.6. In the main phase, an annotated (verified) GPU program and k are given as input. If k is not positive then we report an error. Otherwise we go to the checking phase. This phase checks whether it is possible to unroll the loop k times. Thus, we statically calculate the number of loop iterations, by counting how many times the condition (cond(i)) holds starting from either a (as the lowerbound of i) or b (as the upperbound of i), depending on the operation of upd(i). This is always possible since in addition to cond(i) and upd(i), values of a and b are known (e.g., they are specified in the annotations). We define a new variable *ctr* for calculating the number of loop iterations. If k is greater than the total number of loop iterations at the end of the checking phase, then we report an error. Otherwise we go to the updating phase, in which we update either a or b according to the operation in upd(i). If the operation is addition or multiplication, then the loop variable i (in the unoptimized program) goes from lower bound a to upper bound b. That means, after unrolling, a should be updated according to the constant c from the update expression and k. If the operation is subtraction or division, i goes from b to a. Thus, after unrolling, b should be updated. After the updating phase there is a pre-processing phase where we remove any variable declarations (but we keep initializations) inside the loop and declare them outside the loop. Because, after unrolling, those declarations would be repeated in each unrolled part, which causes errors in the program. When the pre-processing phase is finished we unroll the loop k times and generate the optimized verifiable program as output.

[2] If c was negative, for the multiplication and division, i would oscillate between positive and negative values and hence would not always be useful as array index. Hence we consider c to be positive.

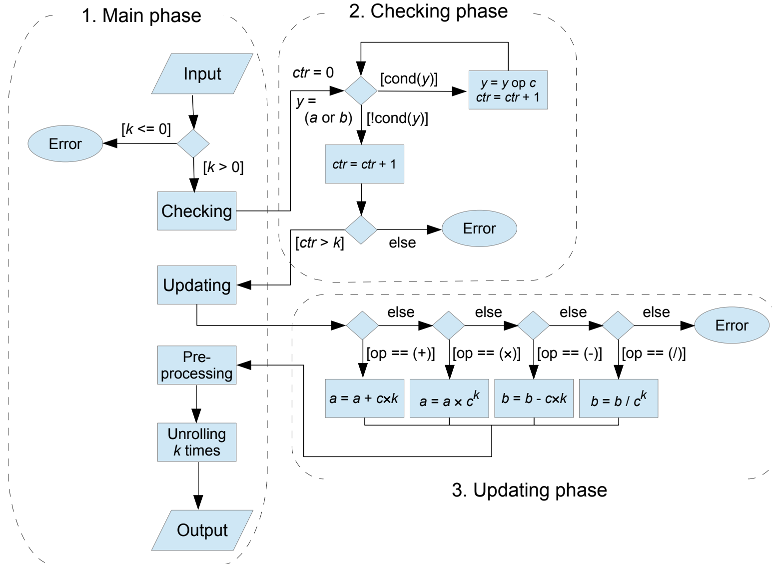

Figure 7.6: The general procedure of loop unrolling optimization.

7.3.2 Iteration Merging

Iteration merging is another optimization technique that is applicable to both GPU and CPU programs[3]. In this optimization, we merge some iterations of the loop into one iteration. Therefore, there are fewer iterations in the optimized program, which leads to fewer comparison operations in the loop. Iteration merging can have a positive performance impact; see [MRBS10, RKBA+13, SIM+10] for the effectiveness of this optimization on GPU programs. In this optimization, an unoptimized verified GPU program (e.g., Figure 7.5) and m as the number of loop iterations to be merged are given as input. Figure 7.7 shows a concrete example of merging a loop three times[4] and Figure 7.8 illustrates the automated procedure of iteration merging while preserving the provability.

As program transformation, first we statically determine the (constant) number of loop iterations (I). Then if I is not a multiple of m, first we unroll the loop I % m times as in the previous section, excluding the checking phase (lines 5-6 and 8-9 in Figure 7.7 (right)). Then we merge m iterations of the loop into one. That means we duplicate the body of the loop m times in a row (lines 17-18, 20-21 and 23-24 in Figure 7.7 (right)).

[3] Iteration merging is also referred to as loop unrolling/vectorization in the literature.

[4] To focus on the main ideas of iteration merging, we omit the assertions that are added after each unrolling part in Figure 7.7.

```
1  void Host(int[] array, int size){
2    par Kernel(tid = 0..size){
3      int i = 1;
4      /*@ inv i ≥ 1 && i ≤ 38;
5          inv Inv(i); @*/
6      loop(i < 20){
7        array[tid] = array[tid] + i;
8        i = i × 2; }
9    }
10 }
```

⇒

```
1  void Host(int[] array, int size){
2    par Kernel(tid = 0..size){
3      //@ ghost int g = 0;
4      int i = 1;
5      array[tid] = array[tid] + i;
6      i = i × 2;
7      //@ ghost g = g + 1;
8      array[tid] = array[tid] + i;
9      i = i × 2;
10     //@ ghost g = g + 1;
11     /*@ inv i ≥ 4 && i ≤ 38;
12         inv g ≥ 2 && g ≤ 5;
13         inv i == pow(2,g);
14         inv g % 3 == 2;
15         inv Inv(i); @*/
16     loop(i < 20){
17       array[tid] = array[tid] + i;
18       i = i × 2;
19       //@ ghost g = g + 1;
20       array[tid] = array[tid] + i;
21       i = i × 2;
22       //@ ghost g = g + 1;
23       array[tid] = array[tid] + i;
24       i = i × 2;
25       //@ ghost g = g + 1; }
26   }
27 }
```

Figure 7.7: An example of merging 3 iterations.

To reverify the optimized program, we should consider a subtle problem. In the original program (Figure 7.7 (left)) the upper bound for i is specified to be 38, because for static program verification, the largest possible value for i where the loop will execute is 19 (thus 38 after the loop body). However, during the program execution, i will never be more than 16 when the loop body is executed, and thus the actual upper bound on i is 32. The situation is more complex in the optimized program. In Figure 7.7, we merge three iterations, hence three update statements (i.e., $i = i \times 2$). That means we cannot establish the invariant for the upper bound in the optimized program (line 11), as 19 (statically worst case) updated thrice is $19 \times 2 \times 2 \times 2 = 152$ which is greater than 38.

To tackle this problem, we add a ghost variable to keep track of the number of loop iterations, and generate more loop invariants. Concretely, we declare a ghost variable g outside the loop which initially is zero. Then this ghost variable is incremented inside the loop. We add the two statements (i.e., `int` $g = 0$ and $g = g + 1$) to the original program to be considered in the unrolling and merging parts

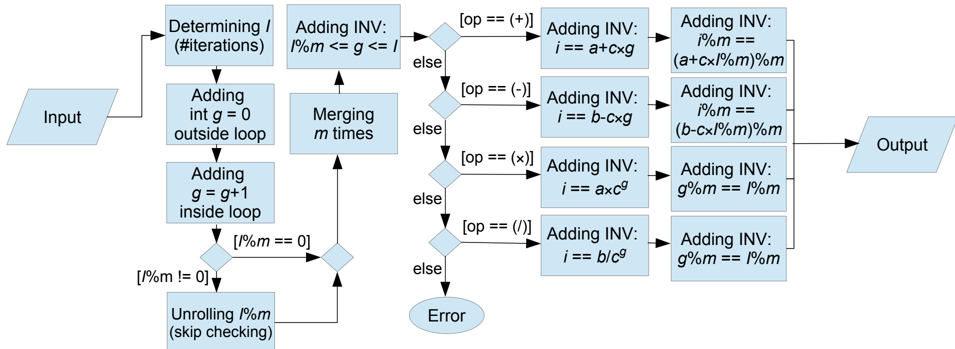

Figure 7.8: The general procedure of iteration merging optimization.

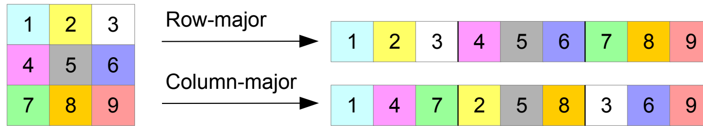

Figure 7.9: Row-major and column-major representation of a matrix.

(lines 3, 7,10 and 19, 22, 25 in Figure 7.7 (right)). Then, we add three invariants (lines 12-14 in Figure 7.7 (right)), which constrain the values of g and how this is related to the value of i. By adding these invariants, we establish the invariant for the upper bound of i as in the original program. Figure 7.8 illustrates the general procedure for all operators in $\texttt{upd}(i)$.

7.3.3 Matrix Linearization

Matrix linearization is an optimization where we transform two-dimensional arrays into one dimension. Since the physical memory address is linear (i.e., one-dimensional), this optimization technique might improve memory access patterns; e.g., see [BG08, CKG14, SHGDM17] for the impact of matrix linearization on GPU programs.

There are two standard techniques to linearize a matrix: 1) a row-major representation where we transform a matrix into an array by storing all elements row by row and 2) a column-major representation where the elements are stored column by column (Figure 7.9).

Figure 7.10 shows an example of how to automatically linearize a matrix with column-major access while preserving the provability of the GPU program. In

```
/*@
context_everywhere mat.length ==
                        M×N;
context (\forall* int i; 0 ≤ i &&
   i < M; (\forall* int j; 0 ≤ j &&
   j < N; Perm(mat[i][j], 1)));
ens (\forall* int i; 0 ≤ i && i < M;
     (\forall* int j; 0 ≤ j && j < N;
     mat[i][j] == \old(mat[i][j])+1));
@*/
void Host(int[][] mat,int M,int N){
  /*@
  context Perm(mat[tidX][tidY], 1);
  ens mat[tidX][tidY] ==
       \old(mat[tidX][tidY])+1;
  @*/
  par Kernel(tidX=0..M, tidY=0..N){
    mat[tidX][tidY] =
    mat[tidX][tidY]+1;
  }
}
```

⇒

```
/*@
context_everywhere mat.length ==
                        M×N;
context (\forall* int i; 0 ≤ i &&
   i < M; (\forall* int j; 0 ≤ j &&
   j < N; Perm(mat[M×j+i], 1)));
ens (\forall* int i; 0 ≤ i && i < M;
     (\forall* int j; 0 ≤ j && j < N;
mat[M×j+i] == \old(mat[M×j+i])+1));
@*/
void Host(int[] mat,int M,int N){
  /*@
  context Perm(mat[M×tidY+tidX], 1);
  ens mat[M×tidY+tidX] ==
       \old(mat[M×tidY+tidX])+1;
  @*/
  par Kernel(tidX=0..M, tidY=0..N){
    mat[M×tidY+tidX] =
    mat[M×tidY+tidX]+1;
  }
}
```

Figure 7.10: An example of linearizing the matrix *mat* with column-major access and dimensions M and N.

this figure, we see that both the code and the annotations need to be transformed (such as line 13 for the annotation and line 18 for the code). Note that the number of threads is the same in the original and the optimized program.

The procedure to automatically linearize a matrix *mat* (M rows, N columns) in a kernel with two-dimensional thread blocks (where each thread has a pair of identifiers $tidX$ and $tidY$) is as follows:

- Change the declaration of *mat* to a declaration of a one-dimensional array with the same name.
- Change all accesses of $mat[tidX][tidY]$ to $mat[N \times tidX + tidY]$ (for row-major access) or $mat[M \times tidY + tidX]$ (for column-major access) in both code and annotations.

7.3.4 Data Prefetching

Suppose there is a verified GPU program where each thread accesses an array location in global memory multiple times. In this optimization, we prefetch the values of those locations that are in global memory into registers which are local to each thread. Therefore instead of multiple accesses to the high latency global

```
1  void Host(int[] a, int[] b, int N){
2    par Kernel(tid = 0..a.length){
3      int counter = 0;
4      /*@
5      inv Inv(i);
6      inv a[tid] ==
7      \old(a[tid])+counter×(b[tid]+1);
8      @*/
9      while(counter < N){
10       // 1 read and 1 write
11       a[tid] = b[tid]+1;
12       counter = counter+1;
13     }// In total N reads and N writes
14   }
15 }
```

⇒

```
1  void Host(int[] a, int[] b, int N){
2    par Kernel(tid = 0..a.length){
3      // 2 reads
4      int a_reg = a[tid];
5      int b_reg = b[tid];
6      int counter = 0;
7      /*@
8      inv Inv(i);
9      inv a_reg ==
10     \old(a[tid])+counter×(b_reg+1);
11     @*/
12     while(counter < N){
13       a_reg = b_reg+1;
14       counter = counter+1;
15     }
16     a[tid] = a_reg; // 1 write
17   }
18 }
```

Figure 7.11: An example of prefetching $a[tid]$ and $b[tid]$.

memory, we benefit from low-latency registers. Data prefetching can have a positive performance impact; see [ALKR20, USQ12, YXKZ10]. Figure 7.11 shows an example of data prefetching. In this figure, the data prefetching optimization reduces N reads and writes to global memory into 2 reads and 1 write.

The procedure to automatically prefetch the value at index tid of any array a while preserving the provability is as follows:

- Declare a variable once (in register) and assign $a[tid]$ to it at the beginning of the kernel.
- For each access to $a[tid]$ in the kernel code, replace it by the new variable.
- For each access to $a[tid]$ in the annotations inside the kernel, replace it by the new variable, except if $a[tid]$ is used in an `\old` expression or as part of a permission predicate.
- If $a[tid]$ is written to, assign the final value of the variable to $a[tid]$.

The `\old` expression and permission predicates are not transformed because (1) the `\old` expressions refer to the values of the locations before the function, which is not defined for registers; (2) the permission predicates can only be used for heap locations and not for registers.

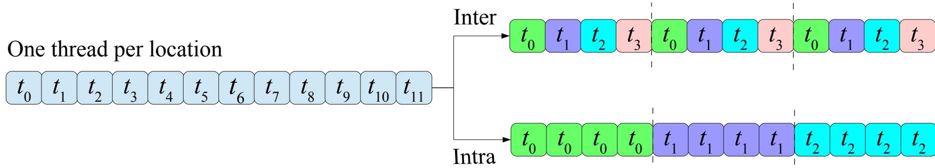

Figure 7.12: Inter- and intra-tiling of an array as $T = 12$, $N = 4$ and $\lceil T/N \rceil = 3$.

```
1 void Host(int[] a, int T){
2   /*@ // Preconditions related to permissions and functional correctness
3       req prePerm(a[tid]) ** preFunc(a[tid]);
4       // Postconditions related to permissions and functional correctness
5       ens postPerm(a[tid]) ** postFunc(a[tid]); @*/
6   par Kernel(tid = 0..T)
7   { body(a[tid]); }
8 }
```

Figure 7.13: A general unoptimized GPU program to apply for tiling.

7.3.5 Tiling

Tiling is another well-known optimization technique for GPU programs. It increases the workload of the threads to fully utilize GPU resources by assigning more data to each thread. Concretely, we assume there are T threads and a one-dimensional array of size T in the unoptimized GPU program where each thread is responsible for one location in that array (Figure 7.13). To apply the optimization, we first divide the array into $\lceil T/N \rceil$ chunks, each of size N ($1 \leq N \leq T$)[5]. There are two different ways to create and assign threads to array cells (as in Figure 7.12):

- *Inter-Tiling* We define N threads and assign them to one specific location in each chunk. That means each thread serially iterates over all chunks to be responsible for a specific location in each chunk.
- *Intra-Tiling* We define $\lceil T/N \rceil$ threads and assign one thread to one chunk (i.e., 1-to-1 mapping) to serially iterate over all cells in that chunk.

Both forms of tiling can have a positive impact on GPU program performance; e.g., see [XKJ09, RPRG17, HSRN+19, KKRS13] for the impact of this optimization.

Figure 7.14 shows the optimized version of Figure 7.13 by applying inter-tiling. Regarding program optimization, two major changes happen: 1) the total number

[5]Since N is in the range $1 \leq N \leq T$, the last chunk might have fewer cells.

```
1  void Host(int[] a, int T){
2    /*@ req (\forall* int i; 0 ≤ i && i < ceiling(T, N) && tid+i×N < T;
3              pre(a[tid+i×N]));
4        ens (\forall* int i; 0 ≤ i && i < ceiling(T, N) && tid+i×N < T;
5              post(a[tid+i×N])); @*/
6    par Kernel(tid = 0..N)
7    {
8      int j = 0;
9      /*@ inv j ≥ 0 && j ≤ ceiling(T, N);
10         inv (\forall* int i; 0 ≤ i && i < ceiling(T, N) && tid+i×N < T;
11              prePerm(a[tid+i×N]));
12         inv (\forall int i; j ≤ i && i < ceiling(T, N) && tid+i×N < T;
13              preFunc(a[tid+i×N]));
14         inv (\forall* int i; 0 ≤ i && i < j && tid+i×N < T;
15              postFunc(a[tid+i×N])); @*/
16     loop(tid+j×N < T){
17       body(a[tid+j×N]);
18       j = j + 1;
19     }
20   }
21 }
```

Figure 7.14: Optimized version of the GPU program of Figure 7.13 after applying inter-tiling.

of threads has reduced (line 6), and 2) the body is encapsulated inside a loop (lines 16-19). As mentioned, in inter-tiling, we define N threads instead of T. The number of chunks is indicated by the function `ceiling`(T, N). Each thread in the newly added loop iterates over all chunks (in the range 0 to `ceiling`(T, N)-1) to be responsible for a specific location. This happens by the loop variable j and the loop condition $tid+j\times N < T$. This means, each thread *tid* can access its own location at index *tid* in each chunk. To preserve verifiability, we add invariants to the loop (lines 9-17). Therefore, we specify:

- The boundaries of the loop variable j, which iterates over all chunks.
- A permission-related invariant for each thread in each chunk (line 10). This comes from the precondition of the kernel and is quantified over all chunks.
- An invariant to indicate functional properties of the locations that have not yet been updated by each thread in the body of the loop (line 12). This comes from the functional property as the precondition of the kernel and is quantified over all chunks.
- An invariant to specify how each thread updates the array in each chunk (line 14). This comes from the functional property as the postcondition of the kernel and is quantified over all chunks.

```
1  void Host(int[] a, int T){
2    /*@ (\forall* int i; tid×N ≤ i && i < (tid+1)×N && i < T; pre(a[i]));
3        (\forall* int i; tid×N ≤ i && i < (tid+1)×N && i < T; post(a[i])); @*/
4    par Kernel(tid = 0..ceiling(T, N))
5    {
6      int j = tid×N;
7      /*@ inv j ≥ tid×N && j ≤ (tid+1)×N && j ≤ T;
8          inv (\forall* int i; tid×N ≤ i && i < (tid+1)×N && i < T;
9               prePerm(a[i]));
10         inv (\forall int i; j ≤ i && i < (tid+1)×N && i < T;
11              preFunc(a[i]));
12         inv (\forall* int i; tid×N ≤ i && i < j; postFunc(a[i])); @*/
13     loop(j < (tid+1)×N && j < T){
14       body(a[j]);
15       j = j + 1;
16     }
17   }
18 }
```

Figure 7.15: Optimized GPU program of Figure 7.13 by applying intra-tiling.

Moreover, we modify the specification of the kernel (lines 2-5). Note that we have the condition $tid + j \times N < T$ in all universally quantified invariants, because the last chunk might have fewer cells than N. We quantified the pre- and postcondition of the kernel over the chunks in the same way as the invariants.

Figure 7.15 shows the optimized version of Figure 7.12 by applying intra-tiling. In essence it is similar to inter-tiling, but there are two differences: 1) the total number of threads is `ceiling`$(T,\ N)$ (line 4), and 2) each thread in the loop iterates over cells within its own chunk. Therefore, we have different conditions in the loop as well as in the quantified invariants.

Above, each thread is assigned to one cell. This can easily be generalized to have each thread assigned to one or more consecutive cells (i.e., a task). A similar procedure can be applied as long as the tasks do not overlap, i.e., each cell is assigned to at most one thread.

7.3.6 Kernel Fusion

Kernel fusion is a GPU optimization where we merge two or more consecutive kernels into one. It increases the potential to use thread-local registers to store intermediate results and can lead to less power consumption. See [WLY10, WM14, FMFM15, ATB+15, WDW+12] for the impact of kernel fusion on GPU programs.

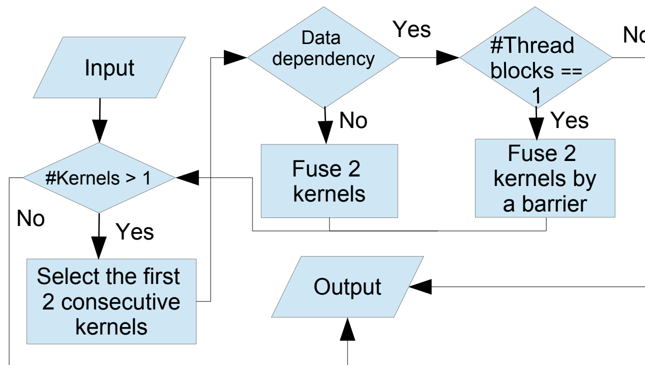

Figure 7.16: The general high-level procedure of kernel fusion optimization.

Figure 7.16 illustrates a high-level procedure of fusing kernels. The figure provides a generalized procedure to fuse an *arbitrary number* of consecutive kernels while considering *data dependency* between them. The idea is to fuse them by repeatedly fusing the first two kernels (i.e., kernel reduction). In each iteration, if there is no data dependency between the two kernels, we safely fuse them. Else if there is only one thread block then we fuse the two kernels by inserting a barrier between the bodies, else fusion fails. That means the procedure terminates in two situations; 1) there are no more kernels to be fused, or 2) there is a data dependency between the two kernels and there is more than one thread block in the kernels. In case 2, kernel fusion is not possible and we should keep the two kernels unfused.

A benefit of this approach is that it only considers two kernels at a time. In this way, it can be determined whether a barrier is necessary between two specific kernels, and we do not miss any possible fusion optimization. Another benefit of this approach is that when a data dependency between two kernels P and $P+1$ ($1 < P <$ #kernels-1) is detected, the output of the approach is the fusion of the first P kernels, and the remaining unfused kernels after P. This allows the user to not only find out that there is a data dependency between P and $P+1$, but also to obtain fused kernels where possible.

There are multiple challenges in this transformation: (1) how to detect data dependency between two kernels? (2) how to collect the pre- and postconditions for the fused kernel? and (3) how to deal with permissions so that in the fused kernel the permission for a location does not exceed 1?

The main difficulty in addressing these challenges is that we have to consider many different possible scenarios. Fortunately, we can use the information from the contracts of the two kernels. The permission patterns in the contracts indicate for

Algorithm 7.1 Kernel fusion procedure for collecting precondition permissions.

```
 1: Add all precondition permissions related to non-shared arrays (i.e., accessed by only one of the two
    kernels) into the contract of the fused kernel kf.
 2: for each shared array a with a permission postPerm(a[e1], p1) in the postcondition of the first
    kernel k1 and a permission prePerm(a[e2], p2) in the precondition of the second kernel k2 do
 3:     if patterns e1 and e2 are syntactically the same then
 4:         Add pre. of k1 corresponding to postPerm(a[e1], p1) as pre. to kf
 5:         if p1 < p2 then
 6:             Add prePerm(a[e2], p2-p1) as pre. to kf
 7:     else if patterns e1 and e2 are not syntactically the same then
 8:         if accumulated pre. permission in both kernels for each array location ≤ 1 then
 9:             Add pre. of k1 corresp. to postPerm(a[e1], p1) and prePerm(a[e2], p2) as pre. in kf
10:         else if p1 < 1 && p2 < 1 then
11:             Add pre. of k1 corresp. to postPerm(a[e1], p1) with permission p3 and prePerm(a[e2],
12:             p4) as pre. s.t. p3 + p4 = 1
13:         else if p1 = 1 (i.e., write) then                        ▷ Data dependency, add barrier
14:             Add pre. of k1 corresponding to postPerm(a[e1], p1) as pre. to kf
15:         else p2 = 1                                                ▷ Data dependency, add barrier
16:             Add pre. of k1 corresponding to postPerm(a[e1], p1) as pre. to kf
17:             Add prePerm(a[e2], 1-p1) as pre. to kf
```

each thread which locations it reads from and writes to. We provide procedures to separately collect pre- and postconditions related to permissions and to functional correctness. Moreover, we provide a procedure to fuse the bodies of the two kernels possibly including an (annotated) barrier.

Algorithm 7.1 shows the essential steps to collect the preconditions related to permissions for array accesses of the fused kernel. It requires kernels *k1* and *k2* not to lose any permissions, only possibly redistribute them (using a barrier). Furthermore, we assume that a write permission is specified for a kernel only if needed. This means that write permissions are not specified if a kernel only reads from an array. Moreover, each thread accesses at most one cell of array a, and the expressions (i.e., e in the algorithm) used to compute array indices only use constants and thread identifier variables, using standard arithmetic operators (e.g., $2 \times tid + 1$).

The algorithm works as follow. First, we compare the postcondition of *k1* and the precondition of *k2* (line 2) to understand how to add permissions of the preconditions of *k1* and *k2* to the precondition of the fused kernel. Note that `prePerm` and `postPerm` correspond to a permission-related pre- and postcondition, respectively. We use the postcondition of *k1* for this comparison since the permission at the end of *k1* needs to be sufficient to satisfy the precondition of *k2*. Thus, if the index expressions *e1* and *e2* to access an array a are syntactically the same (line 3), then they refer to the same array cell. In that case, we first add the *original* permission from the precondition of *k1* to the precondition of the fused kernel. This corresponds to the permission for $a[e1]$ in the postcondition of *k1* (remember that the latter permission may have been obtained in *k1* after permission redistribution). Second, if *p1* is not sufficient for the precondition of *k2* (line 5), we add additional

```
1  void Host(...){
2    /*@ req Perm(a[tid1], 1);
3    req Perm(b[tid1], 1\2);
4    req tid1 != T-1 ?
5    Perm(c[tid1+1], 1\2):Perm(c[0], 1\2);
6    req 2×tid1 < T ==>
7        Perm(d[2×tid1], 1);
8    ens Perm(a[tid1], 1);
9    ens Perm(b[tid1], 1\2);
10   ens Perm(c[tid1], 1\2);
11   ens 2×tid1 < T ==>
12       Perm(d[2×tid1], 1); @*/
13   par Kernel1(tid1 = 0..T)
14   { body1(tid1); }
15
16   /*@ req Perm(a[tid2], 1\2);
17   req Perm(b[tid2], 1);
18   req tid2 != T-1 ?
19       Perm(c[tid2+1], 1):Perm(c[0], 1);
20   req 2×tid2+1 < T ==>
21       Perm(d[2×tid2+1], 1);
22   ens Perm(a[tid2], 1\2);
23   ens Perm(b[tid2], 1);
24   ens tid2 != T-1 ?
25       Perm(c[tid2+1], 1):Perm(c[0], 1);
26   ens 2×tid2+1 < T ==>
27       Perm(d[2×tid2+1], 1); @*/
28   par Kernel2(tid2 = 0..T)
29   { body2(tid2); }
30 }
```

⇒

```
1  void Host(...){
2    /*@ req Perm(a[tid], 1);
3    req Perm(b[tid1], 1\2);
4    req Perm(b[tid1], 1\2);
5    req tid != T-1 ?
6    Perm(c[tid+1], 1\2):Perm(c[0], 1\2);
7    req tid != T-1 ?
8    Perm(c[tid+1], 1\2):Perm(c[0], 1\2);
9    req 2×tid < T ==>
10       Perm(d[2×tid], 1);
11   req 2×tid+1 < T ==>
12       Perm(d[2×tid+1], 1);@*/
13   par Fused_Kernel(tid = 0..T)
14   {
15     ...
16   }
17 }
```

Figure 7.17: Collecting permission preconditions in different cases when fusing two kernels.

permission to the precondition of the fused kernel to satisfy the precondition of *k2* (line 6).

The remaining different cases in the algorithm correspond to the different edge cases that we should consider when *e1* and *e2* are not syntactically the same. In particular, data dependency happens when the accumulated permission (in both kernels) for one location is greater than 1, and there is at least one write permission. Therefore, we have to distinguish multiple cases: 1) the accumulated permission (in both kernels) for one location does not exceed 1 (line 8), 2) the accumulated permission exceeds 1, but no write permission is involved (line 10), or 3) and 4) at least one write is involved (lines 13 and 15).

In case 1, we collect the preconditions in both kernels (line 9). In case 2, we collect

Algorithm 7.2 Kernel fusion procedure for collecting postcondition permissions.

1: Add all postcondition permissions related to non-shared arrays (i.e., accessed by only one of the two kernels) into the contract of the fused kernel *kf*.
2: **for** each shared array *a* with a permission `postPerm`($a[e1]$, *p1*) in the postcondition of the first kernel *k1* and a permission `prePerm`($a[e2]$, *p2*) in the precondition of the second kernel *k2* **do**
3: **if** patterns *e1* and *e2* are syntactically the same **then**
4: Add post. of *k2* corresponding to `prePerm`($a[e2]$, *p2*) as post. to *kf*
5: **if** $p1 > p2$ **then**
6: Add post. of *k2* corresponding to `prePerm`($a[e2]$, *p2*) with permission *p1*-*p2* to *kf*
7: **else if** patterns *e1* and *e2* are not syntactically the same **then**
8: **if** accumulated pre. permission in both kernels for each array location ≤ 1 **then**
9: Add `postPerm`($a[e1]$, *p1*) and post. of *k2* corresp. to `prePerm`($a[e2]$, *p2*) as post. in *kf*
10: **else if** $p1 < 1$ && $p2 < 1$ **then**
11: Add `postPerm`($a[e1]$, *p3*) and post. of *k2* corresp. to `prePerm`($a[e2]$, *p2*) with permission
12: *p4* as post. s.t. $p3 + p4 = 1$
13: **else if** $p2 = 1$ (i.e., write) **then** ▷ Data dependency, add barrier
14: Add post. of *k2* corresponding to `prePerm`($a[e2]$, *p2*) as post. to *kf*
15: **else** $p1 = 1$ ▷ Data dependency, add barrier
16: Add post. of *k2* corresponding to `prePerm`($a[e2]$, *p2*) as post. to *kf*
17: Add post. of *k2* corresponding to `prePerm`($a[e2]$, *p2*) with perm. 1-*p2* as post. to *kf*

the precondition of both kernels, but we update the amount of permissions in such a way that the permissions for each array location is exactly 1 (line 11). In case case 3, we only collect the precondition of *k1* (line 14). Finally in case 4, we collect the precondition of *k1* and we also collect the precondition of *k2* with updated permission 1-*p1* (line 16). Note that, in the latter two cases, a barrier must be introduced to take care of distributing permissions from the access in *k1* to the access in *k2*.

Example Figure 7.17 shows an example of fusing two kernels. We assume there is one thread block in each kernel. For the purpose of this example, we only represent pre- and postcondition specification related to permissions. There are four shared variables, namely a, b, c and d. To collect permission preconditions in the fused kernel according to Algorithm 7.1, we follow the steps in lines 3-4 for variable a, the steps in lines 3-6 for variable b and the steps in lines 7-9 for variable d. For variable c, we follow the steps in lines 15-17 and there is data dependency. □

Algorithm 7.2 shows the essential steps to collect the postcondition related to permissions for array accesses of the fused kernel. It is in spirit the same as Algorithm 7.1, but there are some differences: 1) lines 3-6 in Algorithm 7.2 are the dualities of lines 3-6 in Algorithm 7.1, 2) in case of data dependency, if *p2* is a write permission (lines 13) we collect the postcondition of the second kernel (i.e., *k2*) in line 14. This is because we know that kernels do not loose permissions, then the permission postcondition of *k2* has write permission. In case *p2* is not a write permission, but *p1* is a write permission (line 15) we collect the postcondition of *k2* (lines 16). Moreover, we add another same postcondition with updated permission 1-*p2* (line 17). This is because the precondition of the fused kernel (i.e., *kf*) has write permission and since kernels cannot loose permissions, the postcondition of

```
1  void Host(...){
2    /*@ req Perm(a[tid1], 1);
3    req Perm(b[tid1], 1\2);
4    req tid1 != T-1 ?
5    Perm(c[tid1+1], 1\2):Perm(c[0], 1\2);
6    req 2×tid1 < T ==>
7        Perm(d[2×tid1], 1);
8    ens Perm(a[tid1], 1);
9    ens Perm(b[tid1], 1\2);
10   ens Perm(c[tid1], 1\2);
11   ens 2×tid1 < T ==>
12       Perm(d[2×tid1], 1); @*/
13   par Kernel1(tid1 = 0..T)
14   { body1(tid1); }
15
16   /*@ req Perm(a[tid2], 1\2);
17   req Perm(b[tid2], 1);
18   req tid2 != T-1 ?
19       Perm(c[tid2+1], 1):Perm(c[0], 1);
20   req 2×tid2+1 < T ==>
21       Perm(d[2×tid2+1], 1);
22   ens Perm(a[tid2], 1\2);
23   ens Perm(b[tid2], 1);
24   ens tid2 != T-1 ?
25       Perm(c[tid2+1], 1):Perm(c[0], 1);
26   ens 2×tid2+1 < T ==>
27       Perm(d[2×tid2+1], 1); @*/
28   par Kernel2(tid2 = 0..T)
29   { body2(tid2); }
30 }
```

⇒

```
1  void Host(...){
2    /*@ ens Perm(a[tid], 1\2);
3    ens Perm(a[tid], 1\2);
4    ens Perm(b[tid], 1);
5    ens 2×tid < T ==>
6        Perm(d[2×tid], 1);
7    ens 2×tid+1 < T ==>
8        Perm(d[2×tid+1], 1);
9    ens tid != T-1 ?
10   Perm(c[tid+1], 1):Perm(c[0], 1); @*/
11   par Fused_Kernel(tid = 0..T)
12   {
13     ...
14   }
15 }
```

Figure 7.18: Collecting permission postconditions in different cases when fusing two kernels.

kf should return write permissions for each array location.

Example Figure 7.18 shows the same example of fusing two kernels, but for collecting permission postconditions. We follow Algorithm 7.2 to collect permission postconditions in the fused kernel for the four shared variables. For variable a, we go through the steps in lines 3-6, for variable b the steps in lines 3-4 and for variable d the steps in lines 7-9. For variable c, we follow the steps in lines 13-14 in Algorithm 7.2. □

Algorithm 7.3 shows how to collect the pre- and postcondition related to functional correctness. First, we add all pre- and postcondition specifications of *k1* and *k2*, which only non-shared variables are specified, into the fused kernel (line 1). Then, we add all precondition related to functional correctness of *k1* into the precondition

Algorithm 7.3 Kernel fusion procedure for collecting pre- and postcondition functional correctness.

1: Add all pre- and post. functional correctness specifications in *k1* and *k2*, where only non-shared arrays are specified, into the contract of the fused kernel *kf*.
2: Add all pre. func. correctness of *k1*, where at least one shared array is specified, as pre. to *kf*
3: Add all post. func. correctness of *k2*, where at least one shared array is specified, as post. to *kf*
4: **for** each (not added yet) pre. func. correctness specification *S2* in *k2* and each post. *S1* in *k1*, where at least one shared array is specified, **do**
5:     **if** (all shared ones in *S2* are data independent) && ((no shared ones in *S2* are written to in *k1*) || (*k1* and *k2* access disjoint locations in the shared arrays)) **then**
6:         Add *S2* as pre. to *kf*
7:     **if** (all shared ones in *S1* are data independent) && ((no shared ones in *S1* are written to in *k2*) || (*k1* and *k2* access disjoint locations in the shared arrays)) **then**
8:         Add *S1* as post. to *kf*

of *kf* and the postcondition (related to functional correctness) of *k2*, where at least one shared array is specified there, into the postcondition of *kf* (lines 2-3). Next, we investigate which precondition (related to functional correctness) of *k2* and postcondition of *k1*, where at least one shared array is specified, can be added to *kf* (lines 4-8). The condition to collect them is that all shared arrays specified in a specification must not be data dependent and, either they are not written to in the other kernel or they have disjoint access patterns in *k1* and *k2* (lines 5 and 7). In this way if the specification is not affected by the other kernel, we correctly add it to the contract of *kf*.

Example Figure 7.19 shows an example of how to collect the contract for functional correctness. In this example we only represent specification for functional correctness of arrays a, b and d. We follow Algorithm 7.3 to add the contract for the three shared variables in the fused kernel. According to lines 2 and 3 in the algorithm, we first add all preconditions of the first kernel and all postconditions of the second kernel into the fused kernel. This is shown in lines 2-4 and lines 7-10 in the fused kernel in Figure 7.19. Then, for the rest of the contract, we follow the steps 4-8 of Algorithm 7.3. Concretely, for each precondition (related to functional correctness) in the second kernel, we check the condition (as in line 5 in Algorithm 7.3) to add it to the fused kernel. In this example, we can only add $2{\times}tid2{+}1 < T \Rightarrow d[2{\times}tid2{+}1] == -1$, because according to the condition, there is no data dependency for d and both kernels access disjoint locations in d. For the two other preconditions in the second kernel, the condition does not hold, hence we do not collect them. The same way, for the postconditions in the first kernel, we only add $2{\times}tid2 < T \Rightarrow d[2{\times}tid2] == 2{\times}tid2$ (according to line 7 in Algorithm 7.3) to the fused kernel and we do not collect the other two postconditions. □

Finally, Algorithm 7.4 shows how to fuse the bodies of the two kernels. We fuse the body of *k1* followed by the body of *k2*. In case there is data dependency between

```
1  void Host(...){
2    /*@ req b[tid1] == 0;
3    req 2×tid1 < T ==>
4        d[2×tid1] == -1;
5    ens a[tid1] == b[tid1];
6    ens b[tid1] == 0;
7    ens 2×tid1 < T ==>
8        d[2×tid1] == 2×tid1; @*/
9    par Kernel1(tid1 = 0..T)
10   { body1(tid1); }
11
12   /*@ req a[tid2] == b[tid2];
13   req b[tid2] == 0;
14   req 2×tid2+1 < T ==>
15       d[2×tid2+1] == -1;
16   ens a[tid2] == 0;
17   ens b[tid2] == tid2;
18   ens 2×tid2+1 < T ==>
19       d[2×tid2+1] == 2×tid2+1; @*/
20   par Kernel2(tid2 = 0..T)
21   { body2(tid2); }
22 }
```

⇒

```
1  void Host(...){
2    /*@ req b[tid] == 0;
3    req 2×tid < T ==>
4        d[2×tid] == -1;
5    req 2×tid+1 < T ==>
6        d[2×tid+1] == -1;
7    ens a[tid] == 0;
8    ens b[tid] == tid;
9    ens 2×tid+1 < T ==>
10       d[2×tid+1] == 2×tid+1;
11   ens 2×tid < T ==>
12       d[2×tid] == 2×tid; @*/
13   par Fused_Kernel(tid = 0..T)
14   {
15     ...
16   }
17 }
```

Figure 7.19: Collecting pre- and postcondition contract related to functional correctness in different cases when fusing two kernels.

Algorithm 7.4 Kernel fusion procedure for fusing the bodies of the two kernels.

1: Add the body of *k1*.
2: **if** there is data dependency for a shared array *a* **then**
3: Add a barrier
4: Add the post. permission related to *a* in *k1* as the precondition of the barrier
5: **if** in case of line 15 in Algorithm 7.1 **then**
6: Add `prePerm`($a[e2]$, 1-$p1$) as the precondition of the barrier
7: Add the pre. permission related to *a* in *k2* as the postcondition of the barrier
8: **if** in case of line 15 in Algorithm 7.2 **then**
9: Add post. of *k2* corresp. to `prePerm`($a[e2]$, $p2$) with perm. 1-$p2$ as the post. of the barrier
10: Add the body of *k2*.

k1 and *k2*, we add an annotated barrier in between. In the annotated barrier, we specify the postcondition of *k1* as barrier's precondition, and the precondition of *k2* as barrier's postcondition. Note that in lines 5 and 8, we should also add the extra permissions that we already added as the pre- and postcondition in *fk* (see line 17 in Algorithm 7.1 and Algorithm 7.2). This extra permission must be specified in the barrier in order to redistribute permissions.

```
1  void Host(...){
2    /*@ req Perm(a[tid1], 1);
3    req Perm(b[tid1], 1\2);
4    req tid1 != T-1 ?
5    Perm(c[tid1+1], 1\2):Perm(c[0], 1\2);
6    req 2×tid1 < T ==>
7        Perm(d[2×tid1], 1);
8    ens Perm(a[tid1], 1);
9    ens Perm(b[tid1], 1\2);
10   ens Perm(c[tid1], 1\2);
11   ens 2×tid1 < T ==>
12       Perm(d[2×tid1], 1); @*/
13   par Kernel1(tid1 = 0..T)
14   { body1(tid1); }
15
16   /*@ req Perm(a[tid2], 1\2);
17   req Perm(b[tid2], 1);
18   req tid2 != T-1 ?
19       Perm(c[tid2+1], 1):Perm(c[0], 1);
20   req 2×tid2+1 < T ==>
21       Perm(d[2×tid2+1], 1);
22   ens Perm(a[tid2], 1\2);
23   ens Perm(b[tid2], 1);
24   ens tid2 != T-1 ?
25       Perm(c[tid2+1], 1):Perm(c[0], 1);
26   ens 2×tid2+1 < T ==>
27       Perm(d[2×tid2+1], 1); @*/
28   par Kernel2(tid2 = 0..T)
29   { body2(tid2); }
30 }
```

⇒

```
1  void Host(...){
2    /*@ req Perm(a[tid], 1);
3    req Perm(b[tid1], 1\2);
4    req Perm(b[tid1], 1\2);
5    req tid != T-1 ?
6    Perm(c[tid+1], 1\2):Perm(c[0], 1\2);
7    req tid != T-1 ?
8    Perm(c[tid+1], 1\2):Perm(c[0], 1\2);
9    req 2×tid < T ==>
10       Perm(d[2×tid], 1);
11   req 2×tid+1 < T ==>
12       Perm(d[2×tid+1], 1);
13   ens Perm(a[tid], 1\2);
14   ens Perm(a[tid], 1\2);
15   ens Perm(b[tid], 1);
16   ens 2×tid < T ==>
17       Perm(d[2×tid], 1);
18   ens 2×tid+1 < T ==>
19       Perm(d[2×tid+1], 1);
20   ens tid != T-1 ?
21   Perm(c[tid+1], 1):Perm(c[0], 1); @*/
22   par Fused_Kernel(tid = 0..T)
23   {
24    body1(tid);
25    /*@ req Perm(c[tid], 1\2);
26    req tid != T-1 ?
27    Perm(c[tid+1],1\2):Perm(c[0],1\2);
28    ens tid2 != T-1 ?
29    Perm(c[tid+1], 1):Perm(c[0], 1);@*/
30    barrier(); // data depen. for c
31    body2(tid);
32   }
33 }
```

Figure 7.20: Fusing two kernels using an annotated barrier.

Example Figure 7.20 shows the same example of fusing two kernels using an (annotated) barrier. We also represent the contract related to permissions to explain how to collect permission specification in the barrier. As there is only one thread block, we can fuse the two kernels by inserting a barrier between the bodies (line 30 in Figure 7.20). We follow the steps in lines 1-7 and line 10 in Algorithm 7.4. Concretely, in the contract of the barrier, we add whatever postcondition for c is in the first kernel as the precondition in the barrier (line 25). The body of the first kernel requires the precondition on line 6 in the fused kernel and redistributes it to line 25 in the contract of the barrier (corresponding to the postcondition of the first kernel on line 10). The other precondition in the barrier

Name	**Placement**	**Annotation**
Loop unrolling	Before annotated loop	gpuopt loop_unroll <iteration variable> <#unrollings>
Iteration merging	Before annotated loop	gpuopt iter_merge <iteration variable> <#mergings>
Matrix linearization	Before annotated kernel	gpuopt matrix_lin <matrix name> <C/R> <#rows> <#columns>
Data prefetching	Before annotated kernel	gpuopt glob_to_reg <array name> <index>
Tiling	Before annotated kernel	gpuopt tile <inter/intra> <stride>
Kernel fusion	Before annotated kernel	gpuopt fuse <#fusions> <thread block dimensions>

Figure 7.21: A list of optimizations and their corresponding annotations in VerCors.

(line 26) comes from the other precondition for c in the fused kernel (line 8). Moreover, we add the precondition for c in the second kernel as the postcondition in the barrier (line 28). In this way, the first body has the required permissions and the barrier redistributes the permissions to be used by the second body. □

7.4 Implementation

This section gives a high-level overview of the implementation of the six optimizations in VerCors. The user specifies the optimizations to apply as an annotation in the program along with a flag when invoking VerCors.

Figure 7.21 shows the annotations for the optimizations. For example, to linearize a matrix we annotate the kernel with gpuopt matrix_lin <matrix name> <C/R> <#rows> <#columns> where C and R indicate column-major and row-major representations, respectively. In addition to these annotations, two flags must be given to VerCors to apply the optimizations. The first flag is -gpuopt followed by the desired optimizations options, for instance loop_unroll, iter_merge, matrix_lin, etc. The second flag is -encoded-gpuopt followed by a filename for the optimized

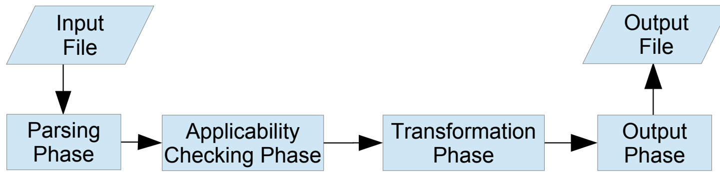

Figure 7.22: The flow chart for optimizing a program in VerCors.

program.

The input program goes through four phases (Figure 7.22): the *parsing* phase, the *applicability checking* phase, the *transformation* phase and the *output* phase. The *parsing phase* transforms the input file into a COL AST, after which the *applicability checking phase* checks if the optimization can be applied. Some optimizations, such as tiling, are always applicable, hence their applicability check always passes. For other optimizations, prerequisites have to be established. Sometimes, a syntactical analysis of the AST suffices. For instance when considering kernel fusion, it must be determined whether there is any data dependency between two selected kernels. When analysis of the AST is not enough, VerCors can be used to perform more complex reasoning. An example of this is loop unrolling. Its prerequisite is that for the loop to be unrollable k times, it is guaranteed that the loop executes at least k times. This prerequisite is encoded as an assertion to be proven by VerCors.

The applicability checking phase exploits the fact that the input program is annotated to determine whether an optimization is applicable. It relies on the fact that VerCors can perform complex reasoning. Moreover, in this way we can distinguish failure due to unsatisfied prerequisites and due to mistakes in the transformation procedure. If the applicability check passes (i.e., the optimization is applicable), the transformation phase is next, otherwise a message is generated that the prerequisites could not be proven.

One of the implementation challenges of kernel fusion is to check for data dependency in the applicability checking phase. To do this for a specific shared array, the function SV is used. Figure 7.23 shows an example of the output of SV. Here, the kernel has $1\backslash 2$ permission for $a[tid{+}1]$ and $1\backslash 3$ permission for $a[0]$ if $tid{+}1$ is out of bounds. SV takes the name of an array and the pre- and postconditions of a kernel (of the form $\texttt{cond}(tid) \Rightarrow \texttt{Perm}(a[\texttt{patt}(tid)], p))$ on line 3 and line 6, and returns a mapping from indices $\texttt{patt}(tid)$ to the permissions p (on the right in Figure 7.23).

If the function SV is executed for two kernels to fuse with the same shared array a, the results $SV_1(a)$ and $SV_2(a)$ can be compared to determine whether there is

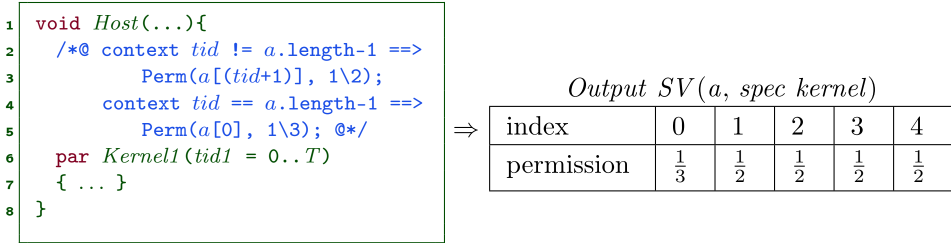

Figure 7.23: Example output of the SV function for array a

data dependency between the two kernels. This comparison is described generally on lines 8-16 in Algorithm 7.1. For each corresponding location in $SV_1(a)$ and $SV_2(a)$, we can determine, for example, whether both permissions combined do not exceed 1 (line 8) or whether the location in *k1* has write permission (line 12).

7.5 Evaluation

This section describes the evaluation of annotation-aware transformation approach which is implemented in VerCors. The goal is first to

Q1 test whether we can apply our approach to GPU programs and then reverify them.

Q2 investigate how long it takes to transform GPU programs and how this affects the verification time.

Q3 apply this technique to some complex verified GPU programs in the first part of the thesis.

7.5.1 Experiment Setup

We evaluate our approach on examples from three different sources. The first source consists of hand-made examples that cover different scenarios for each optimization. The second source is a collection of verified programs from VerCors' example repository[6]. The third source consists of complex case studies that are verified in the first part of the thesis: two parallel prefix sum algorithms, parallel stream compaction and summed-area table algorithms, a variety of sorting algorithms. Moreover, we apply this technique to other big case studies that are already verified in VerCors such as a solution [HJ] to the VerifyThis 2019 challenge 1 [DFH+21] and a Tic-Tac-Toe example [tic] based on [HJ20]. In total, we used

[6]The example repository of VerCors is available at `https://github.com/utwente-fmt/vercors/tree/dev/examples`.

Optimization	Optim. time				Verif. time (orig.)				Verif. time (opt.)			
	min.	**max.**	**avg.**	**med.**	**min.**	**max.**	**avg.**	**med.**	**min.**	**max.**	**avg.**	**med.**
Loop unrolling	0.067	0.238	0.116	0.098	7.6	50.7	18.2	14.3	7.6	57.5	20.8	17.3
Tiling	0.044	0.052	0.048	0.047	16.7	21.5	18.7	18.1	19.3	31.4	24.7	20.8
Kernel fusion	0.099	0.338	0.173	0.137	16.7	54.5	24.6	20.0	14.9	22.3	19.0	19.5
Iteration merging	0.042	0.592	0.152	0.097	6.9	51.0	17.0	12.7	7.3	64.0	20.0	13.8
Matrix linearization	0.011	0.044	0.022	0.017	11.6	16.0	14.3	14.1	11.5	16.8	14.4	15.1
Data prefetching	0.010	0.068	0.051	0.053	9.7	23.0	14.0	13.4	10.4	23.0	13.5	12.7

Figure 7.24: A summary of the optimization and verification times for all six optimizations. All times are in seconds (s).

more than 60 programs, and we applied the optimizations 30 times in the first category, 23 times in the second category and 17 times in the third category (in total 70 experiments). All the experiments were conducted on a MacBook Pro 2020 (macOS 11.3.1) with a 2.0GHz Intel Core i5 CPU. Each experiment was performed ten times, after which the average times, i.e., optimization and verification times, of those executions were recorded for the experiment.

7.5.2 Results & Discussion

Q1 To test whether we can apply our approach to GPU programs and then reverify them, we applied the six optimizations in all 70 experiments and reverify all the resulting programs. All these tests were successful.

Q2 To investigate how long it takes to transform GPU programs and how this affects the verification time, we recorded the transformation time for each optimization applied to all the examples. Figure 7.24 summarizes the best and worst optimization times for the six optimizations. To investigate the impact on the verification time, the table also shows the (best and worst) verification times of the original and optimized programs. The table shows the minimum, maximum, average and median times of all examples. It can be observed that it takes insignificant time to apply each optimization to all the examples. Moreover, the verification time after optimizing generally increases. For loop unrolling, tiling and iteration merging, the verification time increases. This can be attributed to the additional code that is generated. For kernel fusion, the verification time decreases. This is due to verifying fewer kernels. For matrix linearization and data prefetching, the verification time slightly increases. This can be attributed to the linear expressions in matrix linearization and the extra statements to read from/write to the registers in data prefetching.

Q3 To apply this technique to some complex verified GPU programs, we success-

Case	Loop unrolling				Iter. merging				Matrix lin.			Data pref.		
	#	**OT**	**VB**	**VA**	#	**OT**	**VB**	**VA**	**OT**	**VB**	**VA**	**OT**	**VB**	**VA**
BubbleSort	1	0.101	25.4	27.3	4	0.170	29.8	34.1	N/A	N/A	N/A	N/A	N/A	N/A
InsertionSort	1	0.134	25.6	25.8	3	0.225	24.1	28.0	N/A	N/A	N/A	N/A	N/A	N/A
SelectionSort	1	0.107	23.5	25.7	2	0.592	22.8	27.7	N/A	N/A	N/A	N/A	N/A	N/A
TimSort	2	0.216	29.3	38.5	3	0.182	29.1	37.9	N/A	N/A	N/A	N/A	N/A	N/A
Blelloch	1	0.129	50.7	57.5	3	0.355	51.0	64.0	N/A	N/A	N/A	N/A	N/A	N/A
Kogge-Stone	1	0.238	23.0	25.6	2	0.082	21.8	25.6	N/A	N/A	N/A	0.103	23.0	23.0
TicTacToe	3	0.106	19.8	21.0	2	0.076	17.3	19.6	N/A	N/A	N/A	N/A	N/A	N/A
VerifyThis	1	0.144	26.2	28.7	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SATTranspose	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.022	16.0	16.0	N/A	N/A	N/A

Figure 7.25: An overview of optimizing case studies, where # is the unroll factor (for loop unrolling) or the merge factor (for iteration merging), **OT** the time it takes to optimize, **VB** the original verification time (Verification Before) and **VA** the optimized verification time (Verification After). All times are in seconds (s).

fully applied it on the third category with the complex case studies. Figure 7.25 shows the optimization and verification times of applying loop unrolling, iteration merging, matrix linearization and data prefetching to these case studies. Note that in the case studies only these four optimizations could be applied. In the table, `N/A` indicates that the optimization is not applicable to the example.

To further investigate each program, we recorded the number of lines of code for the programs after each optimization, in addition to the verification time. Figures 7.26-7.31 illustrate this information.

Note that the impact of the optimizations on program performance is not discussed in the thesis, as it has been well documented and extensively studied in the literature (e.g., see [MRBS10, ALKR20, XKJ09, WLY10]). If a particular tool is smarter at applying, for instance, kernel fusion, then incorporating the techniques of that tool into VerCors to match the performance is not expected to make annotation transformation infeasible, as the concept of kernel fusion is still respected. Because of this, we do not include an experimental comparison with GPU optimization tools. For our main message, such a comparison would only be relevant if another tool existed that also transforms annotations for code verification.

7.6 Related Work

We categorize the related work into three parts, covering both tools and optimizations.

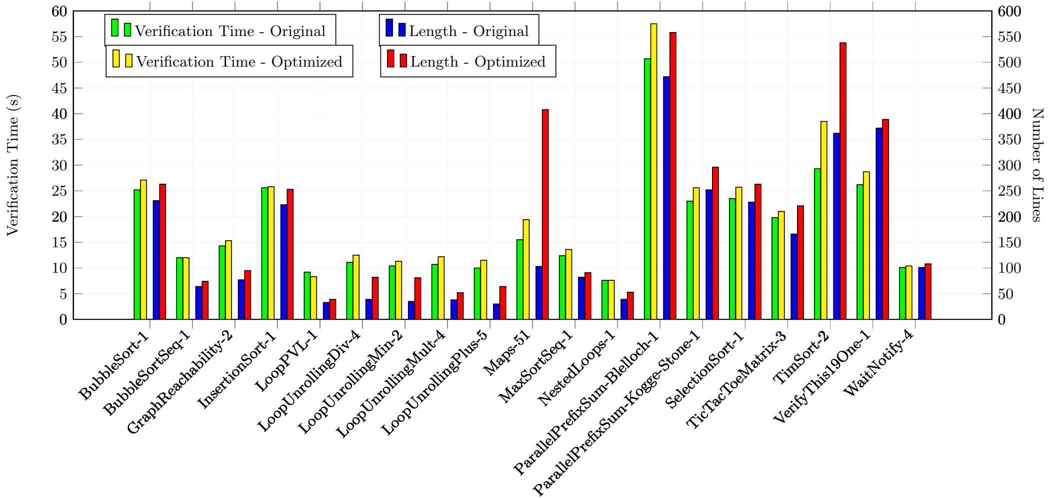

Figure 7.26: The evaluation on the loop unrolling optimization with the number of unrollings as a suffix.

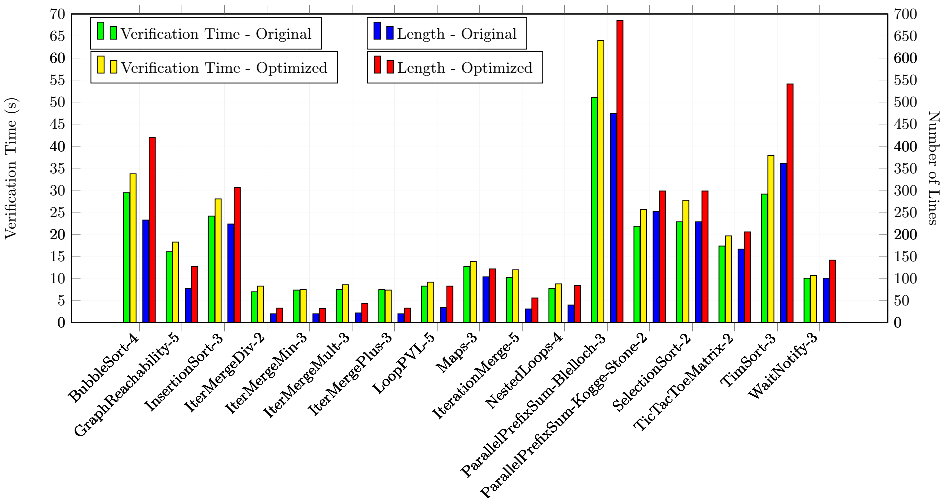

Figure 7.27: The evaluation on the iteration merging optimization with the number of mergings as a suffix.

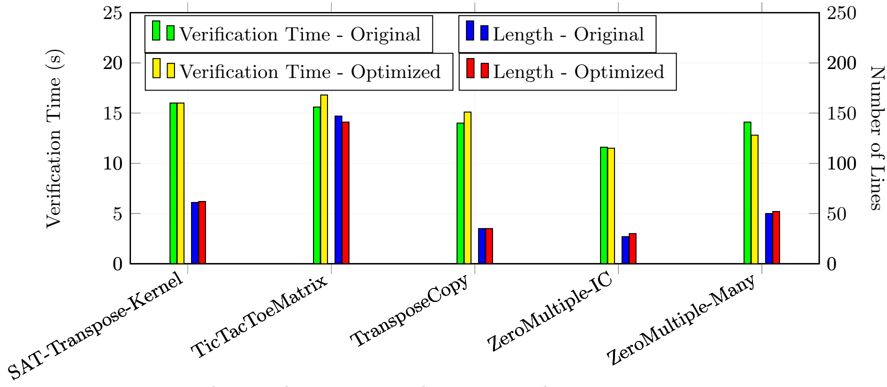

Figure 7.28: The evaluation on the matrix linearization optimization.

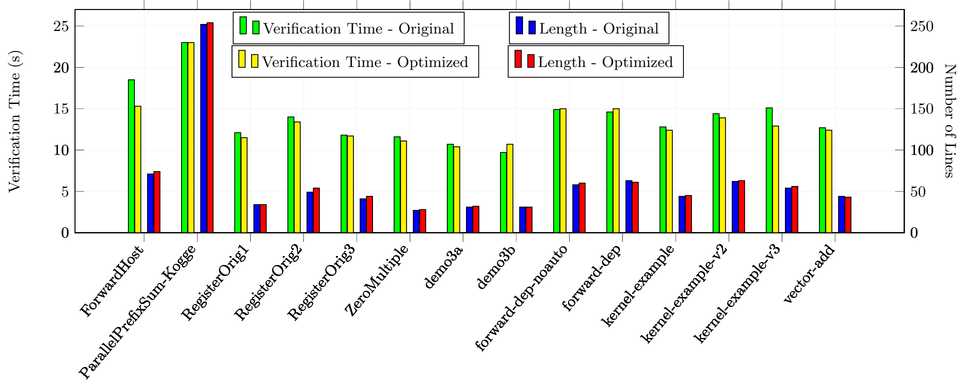

Figure 7.29: The evaluation on the data prefetching optimization.

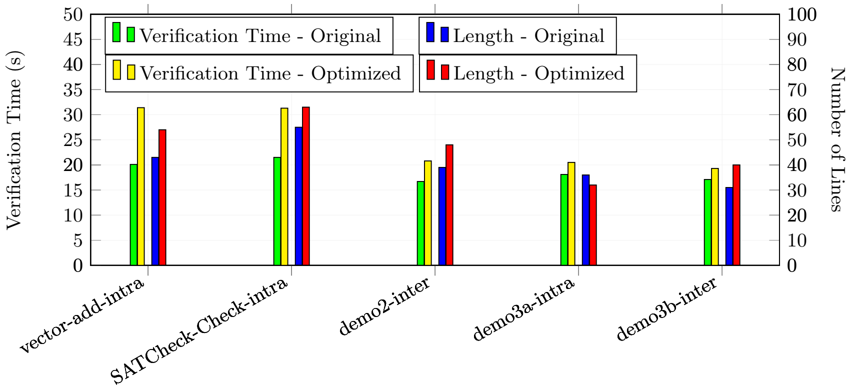

Figure 7.30: The evaluation on the tiling optimization.

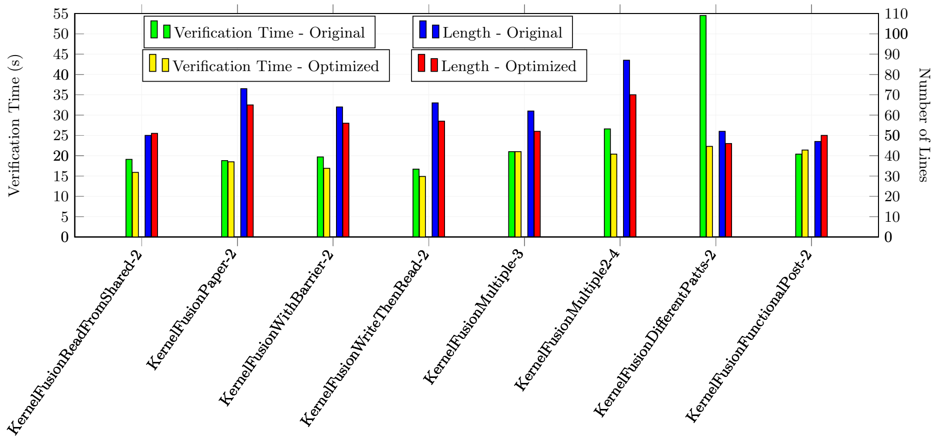

Figure 7.31: The evaluation on the kernel fusion optimization.

Automatic Optimizations without Correctness. One category of related work is about applying automatic optimizations to GPU programs without considering their correctness. Grauer-Gray et al. [GGXS+12] apply optimizations on GPU programs using a high-level directive-based compiler. The focus of the optimizations is mostly related to loops. However, there is no discussion on correctness of the optimized programs. Murthy et al. [MRBS10] develop a semi-automatic, static approach to determine the best configuration for loop unrolling. Their approach differs from ours mainly in two ways. First, they do not reason about the correctness of the optimized program. Second, their approach statically determines the best configurations, while our approach gives the user the flexibility to try different configurations and evaluate which works best. Bell et al. [BG08], Catanzaro et al. [CKG14] and Sundfeld et al. [SHGDM17] discuss the effects of matrix linearization. None of these approaches consider the correctness of the optimized programs, they only focus on the optimization.

Halide [RKBA+13] is a programming language aimed at image and array processing which also targets GPU languages. It makes it simpler to write GPU programs by abstracting over GPU languages, leaving out low-level GPU-specific optimizations. The programmer provides a high-level design of the algorithm and specific optimization to apply on the algorithm. Halide automatically transforms the high-level code into low-level GPU code including the specified optimizations. However, there is no guarantee that the transformations preserve functional correctness of the high-level code.

Kernel tuner [Wer19] is another GPU program optimization tool. In contrast to other tools, kernel tuner only optimizes GPU programs by auto-tuning the configuration for specific hardware. Again, this approach does not provide a framework

for proving properties on GPU programs.

Xu et al. [XKJ09] discuss the tiling optimization in CUDA and how to apply it for different hardware. Rocha et al. [RPRG17] developed a tool to automatically tile stencil computations on GPUs. Winter et al. [HSRN+19] created an adaptive tiling strategy and applied it to operations on sparse matrices. Konstantinidis et al. [KKRS13] developed a scheme for generating programs that use tiling where the tile sizes/chunks can be configured on runtime.

Wang et al. [WLY10] propose the kernel fusion optimization to improve power efficiency and to reduce power consumption. Wahib et al. [WM14] propose a scalable strategy to find an optimal kernel fusion and apply it such that there is a performance gain. Filipovič et al. [FMFM15] optimized BLAS routine calls by applying kernel fusion. Ashari et al. [ATB+15] develop fused kernels for combinations of common BLAS operations. Wu et al. [WDW+12] optimize data warehousing applications by using kernel fusion on relational algebra operators. None of these works consider the correctness of programs.

Ayers et al. [ALKR20] classify memory access patterns for prefetching where the data prefetching optimization is one of the patterns. Yang et al. [YXKZ10] develop a compiler that optimizes memory management and thread-level configurations of GPU programs. They assume that the input is an unoptimized kernel which is functionally correct. Their compiler analyzes the kernel and generates the optimized kernel as output without considering its correctness.

Correctness Proofs for Transformations. Another body of related work focuses on different approaches to preserve provability not specific to GPU programs. COMPCert [Ler06, Ler09] is a formally verified C compiler which preserves semantic equivalence of the source and compiled program, by proving correctness of each transformation in the compilation process. De Putter and Wijs [dPW16] prove the preservation of functional properties over transformations on models of concurrent systems. They prove preservation of model-independent properties. This approach differs from ours as they work on models instead of concrete programs.

Compiler Optimization Correctness. Finally, there is related work that focusses on the compilation of sequential programs, performing transformations from high-level source code to lower-level machine code while preserving the semantics. These approaches neither consider parallelization, nor target different architectures. In GPU programming, the optimizations often need to be applied manually rather than during the compilation process.

Namjoshi and Xu [NX21] use a proof checker to show equivalence between an original WebAssembly program and optimized program. An equivalence proof is

generated based on the transformations. Namjoshi and Singhania [NS16] created a semi-automatic loop optimizer with user-directives. The loops are verified during compilation. For each transformation, semantics are defined to guarantee semantical equivalence to the original program. Namjoshi and Pavlinovic [NP18] focus on recovering from precision loss due to semantics-preserving program transformations and propose systematic approaches to simplify analysis of the transformed program. Finally Gjomemo et al. [GNP+15] help compiler optimizations by supplying high-level information gathered by external static analysis (e.g., Frama-C). This information is used by the compiler for better reasoning.

7.7 Conclusion

In this chapter, we introduced the idea of annotation-aware GPU program optimization. Given an unoptimized, annotated GPU program, we showed how to automatically transform both the code and the annotations, with the goal to preserve the provability of the optimized GPU program. We presented six GPU optimizations, namely loop unrolling, iteration merging, matrix linearization, data prefetching, tiling and kernel fusion. We provided a procedure for each optimization to optimize the code along with its annotation. We implemented the annotation-aware transformation idea as part of the VerCors program verifier and we discussed its design and implementation in detail. Finally, we validated it by verifying a set of examples and reverifying their optimized counterparts. In particular, we successfully applied this technique to some verified complex case studies in the first part the thesis and reverified them.

For future work, there are other optimizations that could be supported, such as data prefetching for all memory patterns as mentioned by Ayers et al. [ALKR20]. Another open question is if and how this approach can be used in program compilation. We also plan to extend this approach to preserve the provability of transpiled code, e.g., CUDA to OpenCL conversions. Moreover, we plan to investigate how this approach can be combined with techniques such as *autotuning* that automatically detect the potential for applying specific optimizations and identify optimal parameter configurations [AKC+18, Wer19].

CHAPTER 8

Conclusions

This thesis contributes to the development of correct optimized GPU programs using deductive verification techniques. GPUs are many-core co-processors and their programming model enables programmers to develop massively parallel programs. GPU programs are widely used in industry and, in particular, in High Performance Computing (HPC) environments. Even though we may achieve better performance by developing GPU programs (in contrast to sequential CPU programs), it may provide risks for program correctness. In the development of GPU programs, typically programmers first implement a naive implementation of an algorithm on a GPU. The goal of this naive version is to establish a base for potential optimizations, as it is easier to validate such an unoptimized program. Then, they apply multiple GPU specific optimizations to efficiently benefit from the underlying resources. As a result, they implement a new version each time according to an optimization, with the goal of outperforming the previous versions. However, each new optimization might introduce non-trivial errors to the program. Therefore the goal of this thesis is to guarantee that the optimized versions of GPU programs remain correct during the development of GPU programs.

This thesis leverages deductive verification techniques to prove data-race freedom and functional correctness of some complicated GPU programs. To show this, we use VerCors, which is a program verifier for concurrent programs based on permission based separation logic. The verification techniques and reasoning patterns that are discussed to verify different parallel algorithms can be beneficial to apply to other algorithms. Moreover, the thesis introduces a technique to automatically apply GPU optimizations to a verified program, and to guarantee that provability of correctness is preserved during such optimizations. As a result, this thesis shows how to develop correct optimized GPU programs. We believe that the thesis demonstrates the feasibility of annotation-aware transformation in practice. This implies that the compilation or transformation process can be extended to include

the extra information that is needed to make sure that the resulting program can be proven correct. The rest of this chapter elaborates more on the contributions of each chapter and then it discusses future work.

8.1 Contributions

In Chapter 3, we prove the correctness of two widely-used parallel prefix sum algorithms on GPUs. In particular, we prove data-race freedom and functional correctness of Blelloch's and Kogge-Stone's prefix sum algorithms on GPUs. These two algorithms are building blocks for other algorithms on GPUs. In the verifications, we show how to benefit from ghost variables to reason about in-place algorithms. In addition, we could reuse the verification effort for proving Blelloch's algorithm to prove Kogge-Stone's algorithm. We also add CUDA support to VerCors and we verify the two prefix sum algorithms implemented in CUDA.

In Chapter 4, we extend the verifications in Chapter 3 to prove two applications of prefix sum. In particular, we prove data-race freedom and functional correctness of stream compaction and summed-area table algorithms on top of the correct prefix sum algorithms. Moreover, we could prove the CUDA version of the stream compaction algorithm. This shows how to reuse a correct sub-function to prove more complicated algorithms. It also shows that it is feasible in practice to prove correctness of complicated parallel algorithms in a modular way by less effort.

In Chapter 5, we prove the correctness of a parallel algorithm to solve the single source shortest path problem. In particular, we prove data-race freedom and functional correctness of the parallel Bellman-Ford algorithm on GPUs. In this verification, we first provide a pen-and-paper proof and then we mechanize the proof in the VerCors verifier. This chapter shows how to automate proofs by contradiction in a program verifier, which is a common approach in proving graph algorithms.

In Chapter 6, we propose a uniform pattern to prove the permutation property of swap-based sorting algorithms. The pattern is general enough to apply it to both parallel and sequential swap-based sorting algorithms. We demonstrate this by applying the proposed technique to well-known parallel and sequential sorting algorithms, namely parallel odd-even transposition sort, and sequential bubble sort, selection sort, insertion sort, quick sort, two in-place merge sorts and TimSort. We use VerCors to prove these sorting algorithms for any arbitrary size of input.

In Chapter 7, we introduce the annotation-aware transformation technique. In this approach, we automatically transform a GPU program along with its annotations such that the optimized program will be still verifiable. Therefore, GPU optimization can be applied to verified programs automatically, while preserving

the provability of their correctness. We show our technique for six well-known GPU optimizations, namely loop unrolling, iteration merging, matrix linearization, data prefetching, tiling and kernel fusion. We discuss the implementation of the annotation-aware transformation in the VerCors verifier. To demonstrate the feasibility of this technique, we apply it to a large number of verified examples. In particular, we apply some of these optimizations to the complicated case studies of the previous chapters in this thesis. Moreover, we illustrate how this technique affects the verification time and the number of lines of codes.

8.2 Future Work

In this section, we discuss several future research directions. In this thesis, we showed how to develop correct optimized GPU programs. The programs are correct in the high-level GPU programming languages. However, the high-level language compiles to further low-level languages to run on hardware. Therefore, during this compilation/transformation, we cannot guarantee that correctness preserves. This is essential to prove that the program is correct in the whole stack, i.e., from high-level code to low-level machine code. As an interesting future work, we can investigate how the annotation-aware transformation technique can be extended to compilation, i.e., how to preserve provability of the (GPU) programs in the compilation from high-level to low-level code. In this way, we can establish a framework to guarantee the correctness of optimized GPU programs executing on hardware.

The challenge in this direction might be that deductive verification techniques are not suitable to reason about low-level code. This might require to design an (abstract) intermediate language for the transformation and to propose a logic to reason about the intermediate representation.

A different, but similar direction is how to use the annotation-aware transformation technique to transform a verified CUDA program into a verified OpenCL program or vice versa. For instance if we plan to reimplement an already verified CUDA version of a case study into OpenCL, how we can automatically generate verifiable OpenCL code from the verified CUDA code.

In addition to high-level to low-level annotation-aware compilation, we provide some concrete future work directions:

Automated specification generation As in the future work, we can automate the process of augmenting source code with annotations. Currently, it requires manual effort to add all the annotations in the program, which is a time-consuming task. We believe a substantial part of the annotations can be generated automatically in VerCors. In particular, the annotations related to permissions are good

candidates to investigate further. It is still a hard problem to generate the full annotations related to loop invariants, but use of artificial intelligence techniques might help in this direction. We also believe that generating annotation for barriers is another interesting starting point to investigate.

Barrier divergence support Another correctness property of GPU programs is barrier divergence-freedom, which we do not address in this thesis. Even though this issue might happen less frequently than data races, it might be a useful extension to our verification approach. A formal operational semantics for barrier divergence freedom is provided in [BCD+15], which is a good place to start investigating how to incorporate that in VerCors.

Extend CUDA and OpenCL in VerCors We made the first step in the verification of CUDA programs in VerCors. We could verify some of the case studies implemented in CUDA, but not all of them. As in the future work, we can extend the capability of VerCors to support more features of CUDA and even OpenCL programs. Therefore, we believe after adding more features to the tool, the remaining algorithms can be verified in CUDA and OpenCL as the main challenges in the verifications are solved using ghost variable as an abstraction.

Floating-point arithmetic support In this thesis, we use integers for all case studies. However, in many GPU applications, floating-point arithmetic is essential. Currently, there is no support for floating-point operations in VerCors. Therefore, reasoning about floating-point numbers in VerCors is an interesting future work. There are several works to reason formally about the floating-point numbers which can be a good starting place for further investigation (e.g., [BTRW15, CIJ+17, ASD+21]).

More case studies The algorithms presented in this thesis for the verification, all are used as building blocks for other algorithms. We already showed a two-layered verification with SC and SAT algorithms. As an extension, one can reuse these verifications and prove further complicated algorithms on top of each other. In this direction, it can be investigated how far we can stack verifications in practice using tool-support, and what will be the limitations to stop us from moving forward.

More annotation-aware optimizations We presented six GPU optimizations and provide procedures for each of them to optimize the program and adapt the annotations according to the optimized code. We can extend this work for other GPU optimizations and provide procedures for them.

APPENDIX **A**

Publications by the Author

- Sebastiaan J.C. Joosten, Wytse Oortwijn, Mohsen Safari, and Marieke Huisman. An exercise in verifying sequential programs with VerCors. In A. Summers, editor, *Formal Techniques for Java-like Programs (FTfJP)*, pages 40–45. ACM, 2018
- Marieke Huisman, Stefan Blom, Saeed Darabi, and Mohsen Safari. Program correctness by transformation. In Tiziana Margaria and Bernhard Steffen, editors, *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Lecture Notes in Computer Science, pages 365–380. Springer, 2018
- Mohsen Safari, Wytse Oortwijn, Sebastiaan Joosten, and Marieke Huisman. Formal verification of parallel prefix sum. In Ritchie Lee, Susmit Jha, Anastasia Mavridou, and Dimitra Giannakopoulou, editors, *NASA Formal Methods*, pages 170–186, Cham, 2020. Springer International Publishing
- Mohsen Safari and Marieke Huisman. Formal verification of parallel stream compaction and summed-area table algorithms. In Violet Ka I Pun, Volker Stolz, and Adenilso Simão, editors, *International Colloquium on Theoretical Aspects of Computing*, Lecture Notes in Computer Science, pages 181–199. Springer, 2020
- Mohsen Safari and Marieke Huisman. A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In Brijesh Dongol and Elena Troubitsyna, editors, *Integrated Formal Methods*, Lecture Notes in Computer Science, pages 257–275. Springer, 2020
- S Blom, S Darabi, M Huisman, and M Safari. Correct program parallelisations. *International Journal on Software Tools for Technology Transfer*, 23(5):741–763, 2021

- Mohsen Safari, Wytse Oortwijn, and Marieke Huisman. Automated verification of the parallel Bellman–Ford algorithm. In Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi, editors, *Static Analysis*, pages 346–358, Cham, 2021. Springer International Publishing
- Mohsen Safari and Marieke Huisman. Formal verification of parallel prefix sum and stream compaction algorithms in CUDA. *Theoretical Computer Science*, 912:81–98, 2022
- Ömer Şakar, Mohsen Safari, Marieke Huisman, and Anton Wijs. Alpinist: An annotation-aware GPU program optimizer. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 332–352, Cham, 2022. Springer International Publishing

Bibliography

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. *Deductive Software Verification – The KeY Book*, volume 10001 of *Lecture Notes in Computer Science*. Springer International Publishing, 2016.

[AD15] Pankhari Agarwal and Maitreyee Dutta. New approach of Bellman Ford algorithm on GPU using compute unified design architecture (CUDA). *International Journal of Computer Applications*, 110(13), 2015.

[ADBH15] Afshin Amighi, Saeed Darabi, Stefan Blom, and Marieke Huisman. Specification and verification of atomic operations in GPGPU programs. In *SEFM 2015 Collocated Workshops*, pages 69–83. Springer, 2015.

[AKC+18] A.H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. A Survey on Compiler Autotuning using Machine Learning. *ACM Computing Surveys*, 51(5):96:1–96:42, 2018.

[ALKR20] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 513–526, 2020.

[ASD+21] Rosa Abbasi, Jonas Schiffl, Eva Darulova, Mattias Ulbrich, and Wolfgang Ahrendt. Deductive verification of floating-point Java programs in KeY. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 242–261. Springer, 2021.

[ATB+15] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. On optimizing machine learning workloads via kernel fusion. *ACM SIGPLAN Notices*, 50(8):173–182, 2015.

[BB16] Federico Busato and Nicola Bombieri. An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2222–2233, 2016.

[BCD+05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[BCD+15] Adam Betts, Nathan Chong, Alastair F Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(3):1–49, 2015.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[BCOP05] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission Accounting in Separation Logic. In *Principles of Programming Languages (POPL)*, pages 259–270, 2005.

[BDHS21] S Blom, S Darabi, M Huisman, and M Safari. Correct program parallelisations. *International Journal on Software Tools for Technology Transfer*, 23(5):741–763, 2021.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, pages 87–90, 1958.

[BFGT15] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Robert E Tarjan. A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms (TALG)*, 12(2):1–22, 2015.

[BG08] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, Citeseer, 2008.

[BHM14] S. Blom, M. Huisman, and M. Mihelčić. Specification and Verification of GPGPU programs. *Science of Computer Programming*, 95:376–388, 2014.

[BK82] Richard P Brent and Hsiang T Kung. A regular layout for parallel adders. *IEEE Computer Architecture Letters*, 31(03):260–264, 1982.

[Ble93] Guy E Blelloch. Prefix sums and their applications. *Synthesis of parallel algorithms. Morgan Kaufmann Publishers Inc., San Francisco*, 1993.

[BMR12] N. Bjørner, K. McMillan, and A. Rybalchenko. Program Verification as Satisfiability Modulo Theories. In *SMT*, 2012.

[BOA09] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the conference on high performance graphics 2009*, pages 159–166, 2009.

[Boy03] J. Boyland. Checking Interference with Fractional Permissions. In R. Cousot, editor, *Static Analysis (SAS)*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[BSSU17] Bernhard Beckert, Jonas Schiffl, Peter H Schmitt, and Mattias Ulbrich. Proving JDK's dual pivot quicksort correct. In *VSTTE*, pages 35–48. Springer, 2017.

[BTRW15] Martin Brain, Cesare Tinelli, Philipp Rümmer, and Thomas Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 160–167. IEEE, 2015.

[CCK11] Peter Collingbourne, Cristian Cadar, and Paul HJ Kelly. Symbolic testing of OpenCL code. In *Haifa Verification Conference*, pages 203–218. Springer, 2011.

[CCL+18] Ran Chen, Cyril Cohen, Jean-Jacques Lévy, Stephan Merz, and Laurent Théry. Formal proofs of Tarjan's algorithm in Why3, Coq, and Isabelle. *arXiv preprint arXiv:1810.11979*, 2018.

[CDK+13] Nathan Chong, Alastair F. Donaldson, Paul H. J. Kelly, Jeroen Ketema, and Shaz Qadeer. Barrier invariants: a shared state abstraction for the analysis of data-dependent GPU kernels. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages*

& Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013, pages 605–622. ACM, 2013.

[CDK14] Nathan Chong, Alastair F Donaldson, and Jeroen Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *ACM SIGPLAN Notices*, volume 49, pages 397–409. ACM, 2014.

[CIJ+17] Sylvain Conchon, Mohamed Iguernlala, Kailiang Ji, Guillaume Melquiond, and Clément Fumex. A three-tier strategy for reasoning about floating-point numbers in SMT. In *International Conference on Computer Aided Verification*, pages 419–435. Springer, 2017.

[CKG14] Bryan Catanzaro, Alexander Keller, and Michael Garland. A decomposition for in-place matrix transposition. *ACM SIGPLAN Notices*, 49(8):193–206, 2014.

[CKL04] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 168–176. Springer, 2004.

[CKSY05] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-Based Predicate Abstraction for ANSI-C. In N. Halbwachs and L. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 570–574. Springer, 2005.

[CLRZ21] Tiago Cogumbreiro, Julien Lange, Dennis Liew Zhen Rong, and Hannah Zicarelli. Checking data-race freedom of GPU kernels, compositionally. In *International Conference on Computer Aided Verification*, pages 403–426. Springer, 2021.

[Coq] The Coq proof assistant. `https://coq.inria.fr`.

[Cro84] Franklin C Crow. Summed-area tables for texture mapping. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 207–212, 1984.

[Daf] Dafny program verifier. `https://github.com/dafny-lang/dafny`.

[DC12] Sandeep Dalal and Rajender Singh Chhillar. Case studies of most common and severe types of software system failure. *International*

Journal of Advanced Research in Computer Science and Software Engineering, 2(8), 2012.

[DĎ88] S Dvořák and B Ďurian. Merging by decomposition revisited. *The Computer Journal*, 31(6):553–556, 1988.

[DDH72] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming.* Academic Press Ltd., 1972.

[DFH+21] Claire Dross, Carlo A Furia, Marieke Huisman, Rosemary Monahan, and Peter Müller. Verifythis 2019: a program verification competition. *International Journal on Software Tools for Technology Transfer*, pages 1–11, 2021.

[dGRdB+15] S. de Gouw, J. Rot, F.S. de Boer, R. Bubel, and R. Hähnle. OpenJDK's java.utils.collection.sort() is broken: The good, the bad and the worst case. In *Computer Aided Verification (CAV)*, volume 9206 of *LNCS*, pages 273–289. Springer, July 2015.

[Dij59] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[Dij76] E. Dijkstra. *A Discipline of Programming.* Englewood Cliffs: Prentice-Hall, 1976.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[dPW16] Sander de Putter and Anton Wijs. Verifying a verifier: on the formal correctness of an LTS transformation verification technique. In *International Conference on Fundamental Approaches to Software Engineering*, pages 383–400. Springer, 2016.

[FGP16] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016.

[Fil11] Jean-Christophe Filliâtre. *Deductive Program Verification.* habilitation, Université de Paris-Sud 11, 2011.

[Fil13] Jean-Christophe Filliâtre. Deductive program verification with Why3 a tutorial, 2013.

[FJ56] Lester R Ford Jr. Network flow theory. Technical report, DTIC Document, 1956.

[Flo67] R. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–31, 1967.

[FM99] Jean-Christophe Filliâtre and Nicolas Magaud. Certification of sorting algorithms in the Coq system. In *TPHOLs*, 1999.

[FMFM15] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing*, 71(10):3934–3957, 2015.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, March 2013.

[GGK06] Alexander Greß, Michael Guthe, and Reinhard Klein. Gpu-based collision detection for deformable parameterized surfaces. In *Computer Graphics Forum*, volume 25, pages 497–506. Wiley Online Library, 2006.

[GGKM06] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, 2006.

[GGXS+12] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a High-Level Language Targeted to GPU Codes. In *Proc. 2012 Innovative Parallel Computing (InPar)*, pages 1–10. IEEE, 2012.

[GJCP19] Armaël Guéneau, Jacques-Henri Jourdan, Arthur Charguéraud, and François Pottier. Formal Proof and Analysis of an Incremental Cycle Detection Algorithm. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:20, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[GKTZ12] Anatoliy Gorbenko, Vyacheslav Kharchenko, Olga Tarasyuk, and Sergiy Zasukha. A study of orbital carrier rocket and spacecraft failures: 2000-2009. *Information & Security*, 28(2):179, 2012.

[GNP+15] Rigel Gjomemo, Kedar S Namjoshi, Phu H Phung, VN Venkatakrishnan, and Lenore D Zuck. From verification to

optimizations. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 300–317. Springer, 2015.

[GT16] Gudmund Grov and Vytautas Tumas. Tactics for the Dafny program verifier. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 36–53. Springer, 2016.

[GZ06] Alexander Greb and Gabriel Zachmann. GPU-ABiSort: Optimal parallel sorting on stream architectures. In *PDP*, pages 10–pp. IEEE, 2006.

[Hab72] AN Habermann. Parallel neighbor sort. *Computer Science Report, Carnegie-Mellon University, Pittsburgh*, 1972.

[HBDS18] Marieke Huisman, Stefan Blom, Saeed Darabi, and Mohsen Safari. Program correctness by transformation. In Tiziana Margaria and Bernhard Steffen, editors, *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Lecture Notes in Computer Science, pages 365–380. Springer, 2018.

[HJ] Marieke Huisman and Sebastiaan Joosten. A solution to VerifyThis 2019 challenge 1. `https://github.com/utwente-fmt/vercors/blob/97c49d6dc1097ded47a5ed53143695ace6904865/examples/verifythis/2019/challenge1.pvl`.

[HJ20] Ruben Hamers and Sung-Shik Jongmans. Safe sessions of channel actions in Clojure: a tour of the discourje project. In *International Symposium on Leveraging Applications of Formal Methods*, pages 489–508. Springer, 2020.

[HN07] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *High performance computing–HiPC 2007*, pages 197–208. Springer, 2007.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM (CACM)*, 12(10):576–580, 1969.

[Hor05] Daniel Horn. Stream reduction operations for GPGPU applications. *GPU gems*, 2(36):573–589, 2005.

[HP14] Gaurav Hajela and Manish Pandey. Parallel implementations for solving shortest path problem using Bellman-Ford. *International Journal of Computer Applications*, 95(15), 2014.

[HSC+05] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. Fast summed-area table generation and its applications. In *Computer Graphics Forum*, volume 24, pages 547–555. Wiley Online Library, 2005.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876, 2007.

[HSRN+19] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.

[HVN09] Pawan Harish, Vibhav Vineet, and PJ Narayanan. Large graph algorithms for massively multithreaded architectures. *International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74*, 2009.

[Isa] Isabelle interactive theorem prover. `http://isabelle.in.tum.de/index.html`.

[IYG+08] F. Ivančić, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based Bounded Model Checking for Software Verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[JKM+14] U. Juhasz, I. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. Technical report, ETH Zürich, 2014.

[JOSH18] Sebastiaan J.C. Joosten, Wytse Oortwijn, Mohsen Safari, and Marieke Huisman. An exercise in verifying sequential programs with VerCors. In A. Summers, editor, *Formal Techniques for Java-like Programs (FTfJP)*, pages 40–45. ACM, 2018.

[JUK+14] In-Kyu Jeong, Jia Uddin, Myeongsu Kang, Cheol-Hong Kim, and Jong-Myon Kim. Accelerating a Bellman–Ford routing algorithm using GPU. In *Frontier and Innovation in Future Computing and Communications*, pages 153–160. Springer, 2014.

[KHB09] Todd Jerome Kosloff, Justin Hensley, and Brian A Barsky. Fast filter spreading and its applications. *EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2009*, 54, 2009.

[Khr08] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008. `http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf`.

[KI13] Kensuke Kojima and Atsushi Igarashi. A Hoare logic for SIMT programs. In *Asian Symposium on Programming Languages and Systems*, pages 58–73. Springer, 2013.

[KI17] Kensuke Kojima and Atsushi Igarashi. A Hoare logic for GPU kernels. *ACM Transactions on Computational Logic (TOCL)*, 18(1):1–43, 2017.

[KK04] Pok-Son Kim and Arne Kutzner. Stable minimum storage merging by symmetric comparisons. In *European Symposium on Algorithms*, pages 714–723. Springer, 2004.

[KKP+15] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing (FAOC)*, 27(3):573–609, 2015.

[KKRS13] Athanasios Konstantinidis, Paul HJ Kelly, J Ramanujam, and Ponnuswamy Sadayappan. Parametric GPU code generation for affine loop programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 136–151. Springer, 2013.

[KMT11] Sumit Kumar, Alok Misra, and Raghvendra Singh Tomar. A modified parallel approach to single source shortest path problem for massively dense graphs using CUDA. In *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on*, pages 635–639. IEEE, 2011.

[KNI14] Akihiko Kasagi, Koji Nakano, and Yasuaki Ito. Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with GPU implementations. In *2014 43rd International Conference on Parallel Processing*, pages 251–260. IEEE, 2014.

[KS73] Peter M Kogge and Harold S Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE transactions on computers*, 100(8):786–793, 1973.

[KT06] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[KVGB14] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. CuSha: vertex-centric graph processing on GPUs.

In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 239–252. ACM, 2014.

[KW05] Peter Kipfer and Rüdiger Westermann. Improved GPU sorting. *GPU gems*, 2:733–746, 2005.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54, 2006.

[Ler09] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[LG10] Guodong Li and Ganesh Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *SIGSOFT FSE 2010, Santa Fe, NM, USA*, pages 187–196. ACM, 2010.

[LLF+96] J. Lions, L. Luebeck, J. Fauquembergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. Ariane 5, Flight 501 Failure, Report by the Inquiry Board, 1996.

[LLG14] Peng Li, Guodong Li, and Ganesh Gopalakrishnan. Practical symbolic race checking of GPU programs. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 179–190. IEEE, 2014.

[LLS+12] Guodong Li, Peng Li, Geof Sawaya, Ganesh Gopalakrishnan, Indradeep Ghosh, and Sreeranga P Rajan. GKLEE: concolic verification and test generation for GPUs. In *ACM SIGPLAN Notices*, volume 47, pages 215–224. ACM, 2012.

[LLVDP+11] Alfons Laarman, Rom Langerak, Jaco Van De Pol, Michael Weber, and Anton Wijs. Multi-core nested depth-first search. In *International Symposium on Automated Technology for Verification and Analysis*, pages 321–335. Springer, 2011.

[LN15] P. Lammich and R. Neumann. A Framework for Verifying Depth-First Search Algorithms. In *CPP*, pages 137–146. ACM, 2015.

[LQ08] S. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification Using SMT Solvers. In *Principles of Programming Languages (POPL)*, pages 171–182. ACM, 2008.

[LT93] N. Leveson and C. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, 1993.

[LW19] Peter Lammich and Simon Wimmer. IMP2-simple program verification in Isabelle/HOL. *Archive of Formal Proofs*, 2019.

[MG10] Duane G Merrill and Andrew S Grimshaw. Revisiting sorting for GPGPU stream architectures. In *PACT*, pages 545–546, 2010.

[MRBS10] Giridhar Sreenivasa Murthy, Mahesh Ravishankar, Muthu Manikandan Baskaran, and Ponnuswamy Sadayappan. Optimal loop unrolling for GPGPU programs. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2010.

[MSS16] P. Müller, M. Schwerhoff, and A. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In B. Jobstmann and K.R.M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 41–62. Springer, 2016.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008.

[NP18] Kedar S Namjoshi and Zvonimir Pavlinovic. The impact of program transformations on static program analysis. In *International Static Analysis Symposium*, pages 306–325. Springer, 2018.

[NS16] Kedar S Namjoshi and Nimit Singhania. Loopy: Programmable and formally verified loop transformations. In *International Static Analysis Symposium*, pages 383–402. Springer, 2016.

[Nvi19] Nvidia. CUDA-MEMCHECK: User manual (version 10), 2019. `https://developer.nvidia.com/cuda-memcheck`.

[NX21] Kedar S Namjoshi and Anton Xue. A Self-certifying Compilation Framework for WebAssembly. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 127–148. Springer, 2021.

[O'H07] P. O'Hearn. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[OHJvdP20] Wytse Oortwijn, Marieke Huisman, Sebastiaan JC Joosten, and Jaco van de Pol. Automated verification of parallel nested DFS.

In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 247–265. Springer, 2020.

[OIH12] Tomohiro Okuyama, Fumihiko Ino, and Kenichi Hagihara. A task parallel algorithm for finding all-pairs shortest paths using the GPU. *International Journal of High Performance Computing and Networking*, 7(2):87–98, 2012.

[Oor19] W. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification*. PhD thesis, University of Twente, Netherlands, 2019.

[Pet02] Tim Peters. Timsort, 2002. `https://bugs.python.org/file4451/timsort.txt`.

[PMS15] James Price and Simon McIntosh-Smith. Oclgrind: An extensible OpenCL device simulator. In *Proceedings of the 3rd International Workshop on OpenCL*, page 12. ACM, 2015.

[PP16] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 1–19. ACM, 2016.

[PR20] Chao Peng and Ajitha Rajan. Automated test generation for OpenCL kernels using fuzzing and constraint solving. In *Proceedings of the 13th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 61–70, 2020.

[Pra95] V. Pratt. Anatomy of the Pentium Bug. In P. Mosses, M. Nielsen, and M. Schwartzbach, editors, *Theory and Practice of Software Development (TAPSOFT)*, pages 97–107. Springer, 1995.

[Rey02] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

[RHVG16] Azalea Raad, Aquinas Hobor, Jules Villard, and Philippa Gardner. Verifying concurrent graph algorithms. In *Asian Symposium on Programming Languages and Systems*, pages 314–334. Springer, 2016.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a

language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[RPRG17] Rodrigo CO Rocha, Alyson D Pereira, Luiz Ramos, and Luís FW Góes. Toast: Automatic tiling for iterative stencil computations on GPUs. *Concurrency and Computation: Practice and Experience*, 29(8):e4053, 2017.

[SA08] Erik Sintorn and Ulf Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.

[Sch16a] Marc Schoolderman. Verification of goroutines using Why3. Master's thesis, Institute for Computing and Information Sciences, RU Nijmegen, 2016.

[Sch16b] M. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution.* PhD thesis, ETH Zürich, 2016.

[SE17] Mohsen Safari and Ali Ebnenasir. Locality-based relaxation: An efficient method for GPU-based computation of shortest paths. In *International Conference on Topics in Theoretical Computer Science*, pages 41–56. Springer, 2017.

[SH20a] Mohsen Safari and Marieke Huisman. Formal verification of parallel stream compaction and summed-area table algorithms. In Violet Ka I Pun, Volker Stolz, and Adenilso Simão, editors, *International Colloquium on Theoretical Aspects of Computing*, Lecture Notes in Computer Science, pages 181–199. Springer, 2020.

[SH20b] Mohsen Safari and Marieke Huisman. A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In Brijesh Dongol and Elena Troubitsyna, editors, *Integrated Formal Methods*, Lecture Notes in Computer Science, pages 257–275. Springer, 2020.

[SH22] Mohsen Safari and Marieke Huisman. Formal verification of parallel prefix sum and stream compaction algorithms in CUDA. *Theoretical Computer Science*, 912:81–98, 2022.

[SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. In *PDP*, pages 1–10. IEEE, 2009.

[SHGDM17] Daniel Sundfeld, Jakob H Havgaard, Jan Gorodkin, and Alba CMA De Melo. CUDA-Sankoff: using GPU to accelerate the pairwise structural RNA alignment. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 295–302. IEEE, 2017.

[SIM+10] Tomoyoshi Shimobaba, Tomoyoshi Ito, Nobuyuki Masuda, Yasuyuki Ichihashi, and Naoki Takada. Fast calculation of computer-generated-hologram on AMD HD5000 series GPU and OpenCL. *Optics express*, 18(10):9955–9960, 2010.

[Skl60] J Sklansky. Conditional-sum addition logic. *IEEE Transactions on Electronic Computers*, 2(EC-9):226–231, 1960.

[SLO06] Shubhabrata Sengupta, Aaron Lefohn, and John Owens. A work-efficient step-efficient prefix sum algorithm, 05 2006.

[SNB15] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 77–87, 2015.

[SOH21] Mohsen Safari, Wytse Oortwijn, and Marieke Huisman. Automated verification of the parallel Bellman–Ford algorithm. In Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi, editors, *Static Analysis*, pages 346–358, Cham, 2021. Springer International Publishing.

[SOJH20] Mohsen Safari, Wytse Oortwijn, Sebastiaan Joosten, and Marieke Huisman. Formal verification of parallel prefix sum. In Ritchie Lee, Susmit Jha, Anastasia Mavridou, and Dimitra Giannakopoulou, editors, *NASA Formal Methods*, pages 170–186, Cham, 2020. Springer International Publishing.

[SS17] Ganesh G Surve and Medha A Shah. Parallel implementation of Bellman-ford algorithm using CUDA architecture. In *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, volume 2, pages 16–22. IEEE, 2017.

[ŞSHW22] Ömer Şakar, Mohsen Safari, Marieke Huisman, and Anton Wijs. Alpinist: An annotation-aware GPU program optimizer. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 332–352, Cham, 2022. Springer International Publishing.

[Sta] Software growth statistics. https://www.statista.com/markets/418/topic/484/software/#overview.

[TGK09] Elena Tushkanova, Alain Giorgetti, and Olga Kouchnarenko. Specifying and Proving a Sorting Algorithm. Technical Report, University of Franche-Comte, October 2009.

[The21a] The CUDA team. Documentation of the CUDA unroll pragma, Accessed Oct. 6, 2021.

[The21b] The Halide team. Documentation of the Halide unroll function, Accessed Oct. 6, 2021.

[tic] The verification of TicTacToe program. https://github.com/utwente-fmt/vercors/blob/0a2fdc24419466c2d3b7a853a2908c37e7a8daa7/examples/session-generate/MatrixGrid.pvl.

[TM11] Asma Tafat and Claude Marché. Binary Heaps Formally Verified in Why3. Research Report RR-7780, INRIA, October 2011.

[TMLT11] Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. Collision-streams: Fast GPU-based collision detection for deformable models. In *Symposium on interactive 3D graphics and games*, pages 63–70, 2011.

[USQ12] Swapneela Unkule, Christopher Shaltz, and Apan Qasem. Automatic restructuring of GPU kernels for exploiting inter-thread data locality. In *International Conference on Compiler Construction*, pages 21–40. Springer, 2012.

[vdHWvdBH20] Lars B van den Haak, Anton Wijs, Mark van den Brand, and Marieke Huisman. Formal methods for GPGPU programming: Is the demand met? In *International Conference on Integrated Formal Methods*, pages 160–177. Springer, 2020.

[vdP15] Jaco C van de Pol. Automated verification of nested DFS. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 181–197. Springer, 2015.

[Vera] The VerCors program verifier. https://github.com/utwente-fmt/vercors.

[Verb] The VeriCUDA program verifier. https://github.com/SoftwareFoundationGroupAtKyotoU/VeriCUDA.

[VME18] Grigoriy Volkov, Mikhail Mandrykin, and Denis Efremov. Lemma functions for Frama-C: C programs as proofs. In *2018 Ivannikov Ispras Open Conference (ISPRAS)*, pages 31–38. IEEE, 2018.

[VSC] A Theory of Bellman–Ford, in Isabelle. `https://www.isa-afp.org/browser_info/current/AFP/Monad_Memo_DP/Bellman_Ford.html` *(accessed on January 2021).*

[vWMBS14] Ben van Werkhoven, Jason Maassen, Henri E Bal, and Frank J Seinstra. Optimizing convolution operations on GPUs using adaptive tiling. *Future Generation Computer Systems*, 30:14–26, 2014.

[WDP+16] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 1–12, 2016.

[WDW+12] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing data warehousing applications for GPUs using kernel fusion/fission. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 2433–2442. IEEE, 2012.

[Wer19] Ben van Werkhoven. Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019.

[WHN18] Simon Wimmer, Shuwei Hu, and Tobias Nipkow. Verified memoization and dynamic programming. In *International Conference on Interactive Theorem Proving*, pages 579–596. Springer, 2018.

[Whya] Why3 gallery of formally verified graph algorithms. `http://toccata.lri.fr/gallery/graph.en.html`.

[Whyb] Why3 program verifier. `http://why3.lri.fr/`.

[Wit16] Alexandra Witze. Software error doomed japanese hitomi spacecraft. *Nature News*, 533(7601):18, 2016.

[WLY10] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350. IEEE, 2010.

[WM14] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound GPU applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202. IEEE, 2014.

[XKJ09] Chang Xu, Steven R Kirk, and Samantha Jenkins. Tiling for performance tuning on different models of GPUs. In *2009 Second International Symposium on Information Science and Engineering*, pages 500–504. IEEE, 2009.

[YXKZ10] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. *ACM Sigplan Notices*, 45(6):86–97, 2010.

[ZH14] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2014.

[ZRQA11] Mai Zheng, Vignesh T Ravi, Feng Qin, and Gagan Agrawal. GRace: a low-overhead mechanism for detecting data races in GPU programs. *ACM SIGPLAN Notices*, 46(8):135–146, 2011.

[ZRQA13] Mai Zheng, Vignesh T Ravi, Feng Qin, and Gagan Agrawal. GMRace: Detecting data races in GPU programs via a low-overhead scheme. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):104–115, 2013.

Samenvatting

Software is een integraal onderdeel van ons dagelijks leven. Bijna alle elektronische apparaten die we dagelijks gebruiken, van smartphones en koffiezetapparaten tot auto's en vliegtuigen, worden bestuurd door software.

Dat betekent dat ons leven sterk afhankelijk is van software, terwijl software door mensen gemaakt is, en mensen maken van nature fouten. Daarom is het maken van *correcte* software een hele belangrijke uitdaging, waarbij we met correctheid bedoelen dat we kunnen garanderen dat software zich gedraagt zoals verwacht. Softwarecorrectheid is met name belangrijke voor kritische systemen, zoals kerncentrales, en medische apparatuur, waarbij softwarefouten ernstige of zelfs levensbedreigende gevolgen kunnen hebben voor mensen of onze omgeving (zie [LT93, Pra95, LLF+96, DC12, GKTZ12, Wit16]).

Naast betrouwbaarheid en correctheid, is ook de executiesnelheid van software belangrijk voor gebruikers, en in het bijzonder voor kritieke systemen waar een vertraagde reactie serieuze schade kan veroorzaken. Om de snelheid van software te verbeteren, kijken we vaak naar nieuwe hardware en architecturen. Volgens de wet van Moore verdubbelt het aantal transistoren op een CPU-chip elke 18 maanden. Daarnaast hebben we de wet van Dennard, die zegt dat als transistoren kleiner worden, hun vermogen constant blijft. Vanwege deze wetten konden chipsfabrikanten de kloksnelheid van chips vergroten zonder dat het elektriciteitsgebruik significant groter werd. Maar vandaag de dag groeit het aantal transistoren niet meer volgens de wet van Moore, terwijl ook Dennard's schaalbaarheidswet niet meer geldt. Daarom wordt er steeds meer gebruik gemaakt van multi-core CPU-architecturen, maar het aantal CPU-cores dat op een machine kan draaien heeft nog beperkingen. Daarom zijn Graphics Processing Units (GPUs) ontwikkeld als co-processoren die de beperkingen van multi-core CPUs addresseren, en tegelijkertijd de snelheidsbeperkingen van data-parallelle programma's verminderen. In tegenstelling tot multi-core CPUs, hebben GPUs honderden of duizenden cores,

die simpeler zijn dan de individuele CPU-cores, maar beter geschikt zijn voor data parallellisme. CUDA en OpenCL zijn twee belangrijke programmeertalen voor GPU-architecturen. Hun programmeermodel ondersteunt parallelle executie, met veel samenwerkende threads, die dezelfde programma-instructies uitvoeren, maar op eigen data (dit heet het Single Instruction Multiple Data (SIMD) paradigma). GPUs worden veel gebruikt in de industrie, omdat ze vaak tot een drastische snelheidsverbetering van (sequentiële of multi-threaded) programma's leiden. Maar helaas komen er ook vaker fouten voor in GPU-programma's, omdat het GPU programmeermodel meer flexibiliteit geeft voor het aansturen van de hardware. Ook de hiërarchische thread en geheugenstructuur maakt het programmeren van een GPU lastiger en foutgevoeliger. Daarom is het correct bewijzen van GPU-programma's uitermate belangrijk. In dit proefschrift onderzoeken we hoe je kunt bewijzen dat een GPU-programma geen data races heeft en functioneel correct is.

Deductieve programmaverificatie kan gebruikt worden om te redeneren over GPU-programma's. Bij een deductieve aanpak worden programma's geannoteerd met pre- en postcondities, invarianten en hulpeigenschappen. Deze worden allemaal geschreven in eerste-orde logica. Vervolgens gebruiken we een bewijssysteem om te verifiëren of de implementatie voldoet aan de specificatie. Het gebruik van deductieve programmaverificatie is uitdagend voor GPU-programma's, omdat het rekening moet houden met de vele verschillende volgordes waarin de verschillende thread uitgevoerd kunnen worden, en met het GPU-specifieke programmeermodel. Daarnaast is het ontwikkelen van GPU-programma's vaak een iteratief proces, waarbij programmeurs eerst een naïeve versie van hun algoritme op de GPU implementeren. We noemen dit de ongeoptimaliseerde versie, omdat de focus ligt op de functionaliteit van de implementatie, in plaats van op de snelheid. Om vervolgens de beste executiesnelheid te bereiken, zullen programmeurs vervolgens diverse incrementele GPU-architectuurspecifieke optimalisaties toepassen, die resulteren in nieuwe versies van de implementaties. Deze optimalisaties worden handmatig of halfautomatisch toegepast op het GPU-programma, voordat deze gecompileerd wordt naar executeerbare code. Hoewel er onderzoek is gedaan naar het automatiseren van dit proces, zijn er weinig garanties dat de correctheid van het programma behouden blijft tijdens dit optimalisatieproces. Dit is wel belangrijk, omdat het belangrijk is dat we kunnen laten zien dat de geoptimaliseerde versie ook correct is, zonder dat we handmatig nieuwe annotaties hoeven toe te voegen.

Daarom adresseert dit proefschrift de volgende twee vragen. Om te beginnen, *hoe kan deductieve verificatie in de praktijk gebruikt worden om de functionele correctheid van niet-triviale GPU-programma aan te tonen?*, en ten tweede *hoe kunnen we automatisch GPU-optimalisaties toepassen op een ongeoptimaliseerd GPU-programma, waarbij we kunnen garanderen dat het geoptimaliseerde programma nog steeds correct is?*

Dit proefschrift bestaat uit twee delen om deze twee vragen te beantwoorden. Het eerste deel laat zien hoe deductieve verificatie aangepast en gebruikt kan worden om te redeneren over niet-triviale GPU-programma's. In het bijzonder bewijzen we de afwezigheid van data races en de functionele correctheid van de implementatie van twee parallelle prefix som algoritmes, een parallel stream compaction algoritme, een parallel summed-area table algoritme, en het parallelle Bellman-Ford kortste pad algoritme. Dit gedeelte van het proefschrift introduceert daarnaast een generieke techniek om de permutatie eigenschap van (GPU-gebaseerde) parallelle of (CPU-gebaseerde) sequentiële sorteeralgoritmen aan te tonen. Deze techniek wordt gebruikt om de permutatie-eigenschap aan te tonen van o.a. parallelle odd-even transposition sort, en sequentiële bubble sort, selection sort, insertion sort, quick sort, twee in place merge-sorts en TimSort.

Het tweede gedeelte van dit proefschrift introduceert een *annotatie-bewuste (source-to-source) transformatie* techniek. Dat wil zeggen dat tijdens de transformatie, niet alleen de programmacode automatisch wordt aangepast, maar ook de corresponderende annotaties. Het doel is deze optimalisatietechniek automatisch toe te passen op een geverifieerd, ongeoptimaliseerd GPU-programma, stap voor stap, zodat de annotaties automatisch getransformeerd worden, en het geoptimaliseerde programma opnieuw geverifieerd kan worden. We laten zien hoe deze techniek te gebruiken is voor zes optimalisatietechnieken, namelijk loops uitrollen, het samenvoegen van iteraties, matrix linearisatie, data vervroegd inlezen in het geheugen, tegelen en kernelfusie. We evalueren deze aanpak op de voorbeeldprogramma's die in het eerste gedeelte van het proefschrift correct bewezen zijn.

Titles in the IPA Dissertation Series since 2019

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow*. Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning*. Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management*. Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers*. Faculty of Science, UvA. 2019-04

S. Kisfaludi-Bak. *ETH-Tight Algorithms for Geometric Network Problems*. Faculty of Mathematics and Computer Science, TU/e. 2019-05

J. Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. Faculty of Science, Mathematics and Computer Science, RU. 2019-06

V. Bloemen. *Strong Connectivity and Shortest Paths for Checking Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-07

T.H.A. Castermans. *Algorithms for Visualization in Digital Humanities*. Faculty of Mathematics and Computer Science, TU/e. 2019-08

W.M. Sonke. *Algorithms for River Network Analysis*. Faculty of Mathematics and Computer Science, TU/e. 2019-09

J.J.G. Meijer. *Efficient Learning and Analysis of System Behavior*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-10

P.R. Griffioen. *A Unit-Aware Matrix Language and its Application in Control and Auditing*. Faculty of Science, UvA. 2019-11

A.A. Sawant. *The impact of API evolution on API consumers and how this can be affected by API producers and language designers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2019-12

W.H.M. Oortwijn. *Deductive Techniques for Model-Based Concurrency Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2019-13

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views*. Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Cosimulation Construction*. Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game*

Design & Enabling Live Programming. Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models.* Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems.* Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking.* Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing.* Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations.* Faculty of Mathematics and Computer Science, TU/e. 2020-09

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security.* Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic.* Faculty of Sciences, Department of Computer Science, VUA. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components.* Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management.* Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS.* Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning.* Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03