Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

PhD Thesis

Improving Performance of Simulations and Heuristic Optimization on GPUs

Dissertation zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Fakultät für Informatik und Mathematik
der Universität des Saarlandes

eingereicht von

Marcel Köster

Saarbrücken, 2023

Date of the Colloquium:
January 30, 2024

Dean:
Prof. Dr. Jürgen Steimle

Members of the Examination Board:

Chair:
Prof. Dr. Bernt Schiele
Reporter:
Prof. Dr. Antonio Krüger
Prof. Dr. Philipp Slusallek
Scientific Assistant:
Dr. Tim Schwartz

Notes on Style
In this thesis, the scientific plural "we" is used instead of "I" to emphasize that this work includes contributions in the form of feedback and suggestions from other researchers and engineers. Additionally, support during the actual realization phases (including implementation and evaluation) of the various research prototypes and production-ready software versions into which the various algorithms were integrated is acknowledged. References to web resources (e.g., links to articles) are given as URLs. Longer URLs have been shortened. It is important to note that most figures, equations, and algorithms in this thesis were published in one of the referenced papers. Figures, equations, and algorithms in this thesis have been adapted to the general style and expanded/updated to provide additional insights. This is made explicitly clear by references to the previously published figures, equations, and algorithms.

Acknowledgment

Now, I would like to express my great gratitude to all people involved supporting me in the preparation and writing of my doctoral thesis. I would especially like to thank Prof. Dr. Antonio Krüger and Prof. Dr. Philipp Slusallek for their outstanding supervision and reviewing of my thesis. In addition, I would like to thank Prof. Dr. Wolfgang Wahlster for his extensive support, advise, feedback, and hints throughout my studies. On this occasion, I would also like to thank Prof. Dr. Bernt Schiele, who chaired my examination board, and Dr. Tim Schwartz, who served as my scientific assistant during my thesis defense.

Furthermore, I would like to express my gratitude to all colleagues at DFKI and Saarland University for their support during my doctoral research phase. In particular, I would like to extend my sincere thanks to Julian Groß for his invaluable collaboration on experiments, idea generation, and co-authorship of papers. Beside the professional support from my colleagues, I also received tremendous encouragement from my friends during my studies.

I am deeply grateful for the unwavering support and encouragement of my parents throughout my academic journey over many years. Their belief in my abilities has been an invaluable source of strength and motivation. I would also like to express my deepest appreciation to my in-laws, whose faith in me has contributed significantly to my confidence and determination. Above all, I owe a profound debt of gratitude to my beloved wife, whose exceptional motivation and belief in me have been the driving force behind my successful completion of this thesis and my focus on research.

For my wife Nurten, my parents Sigrid and Rainer, and my beloved family.

Abstract

Parallelization is a ubiquitous technique for improving runtime performance of algorithms. Although parallelization is generally challenging and often leads to programming bugs, it is a leading method for processing growing amounts of data today. Due to the ongoing trend of exploring the unexplored, known methods are reaching their limits in terms of scalability and thus applicability.

Particularly challenging is the use of graphics processing units (*GPUs*) that require specially optimized algorithms but feature impressive compute power. Unfortunately, the term "optimized" usually refers to newly developed algorithms that exploit the peculiarities of the underlying GPUs or at least follow their specific programming methodologies. The list of tweaked algorithms available for GPUs is already quite long and touch a wide range of domains. These include the well-known fields of massively parallel simulations and solving of optimization problems. Prominent examples in this context include particle simulations of physical processes (like molecular-dynamics simulations) and machine-learning based optimizers. However, existing approaches from these two domains often suffer from severe runtime, memory consumption, and applicability limitations.

In this thesis, we present new approaches for both domains. Our methods considerably outperform current state of the art in terms of runtime and memory consumption. We were able to achieve runtime speedups of up to several orders of magnitude while reducing the amount of memory required compared to existing methods. Regarding applicability, our algorithms are designed to fit seamlessly into existing simulation programs and optimizers. This makes them a particularly valuable contribution to real-world applications as well.

Zusammenfassung

Parallelisierung ist eine allgegenwärtige Technik zur Verbesserung der Laufzeitleistung von Algorithmen. Obwohl Parallelisierung im Allgemeinen eine Herausforderung darstellt und oft zu Programmierfehlern führt, ist sie heute eine führende Methode zur Verarbeitung wachsender Datenmengen. Aufgrund des anhaltenden Trends, das Unerforschte zu erforschen, stoßen die bekannten Methoden an ihre Grenzen der Skalierbarkeit und damit der Anwendbarkeit.

Eine besondere Herausforderung ist der Einsatz von Grafikprozessoren (GPUs), die speziell optimierte Algorithmen erfordern, aber eine beeindruckende Rechenleistung aufweisen. Allerdings bedeuted der Begriff "optimiert" in der Regel neue Algorithmen zu entwickeln, die die Besonderheiten der zugrundeliegenden GPUs ausnutzen oder zumindest deren spezifischen Programmiermethodologien folgen. Die Liste der optimierten Algorithmen, die für GPUs verfügbar sind, ist bereits lang und deckt ein breites Bereichsspektrum ab. Dazu gehören die bekannten Anwendungsfelder der massiv parallelen Simulationen und das Lösen von Optimierungsproblemen. Prominente Beispiele in diesem Zusammenhang sind Partikelsimulationen physikalischer Prozesse (wie z.B. Molekulardynamiksimulationen) und der Einsatz von Optimierern auf der Basis des maschinellen Lernens. Bestehende Ansätze aus diesen beiden Bereichen leiden jedoch oft an schwerwiegenden Einschränkungen in Bezug auf Laufzeit, Speicherverbrauch und Anwendbarkeit.

In dieser Arbeit stellen wir neue Ansätze für beide Domänen vor. Sie übertreffen den aktuellen Stand der Technik in Bezug auf Laufzeit und Speicherverbrauch deutlich. Es konnten signifikante Beschleunigungen von bis zu mehreren Größenordnungen im Vergleich zu bestehenden Verfahren erreicht und dabei zugleich den Speicherbedarf deutlich reduziert werden. Hinsichtlich der Anwendbarkeit sind unsere Algorithmen so konzipiert, dass sie sich nahtlos in bestehende Simulationsprogramme und Optimierer integrieren lassen. Dies macht sie damit auch zu einem besonders wertvollen Beitrag für reale Anwendungen aus der Praxis.

List of Publications

Parts of the work presented in this thesis, including concepts, diagrams, measurements, results, formulas, code listings, algorithms and other text fragments have already been published. The following list summarizes all associated publications on which this thesis is based on (papers highlighted with ⊛ had been awarded a *Best Paper* award):

Conference papers:

[KGK20a] Marcel Köster et al. "High- Performance Simulations on GPUs Using Adaptive Time Steps." In: *20th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2020)*. Springer, 2020

[KGK19a]⊛ Marcel Köster et al. "FANG: Fast and Efficient Successor-State Generation for Heuristic Optimization on GPUs." In: *19th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2019)*. Springer, 2019

[KGK19c]⊛ Marcel Köster et al. "Parallel Tracking and Reconstruction of States in Heuristic Optimization Systems on GPUs." In: *Parallel and Distributed Computing, Applications and Technologies (PDCAT-2019)*. IEEE, 2019

[KK18] Marcel Köster and Antonio Krüger. "Screen Space Particle Selection." In: *Proceedings of the Conference on Computer Graphics & Visual Computing (CGCV-2018)*. The Eurographics Association, 2018

Journal papers:

[KGK20b] Marcel Köster et al. "Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction." In: *International Journal of Parallel Programming* (2020)

[KK16] Marcel Köster and Antonio Krüger. "Adaptive Position-Based Fluids: Improving Performance of Fluid Simulations for Real-Time Applications." In: *International Journal of Computer Graphics & Animation* (2016)

Workshop papers:

[KGK19b] Marcel Köster et al. *Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction.* 12th International Symposium on High-Level Parallel Programming and Applications (HLPP-2019). 2019

[Kös+15] Marcel Köster et al. "Asterodrome: Force-of-Gravity Simulations in an Interactive Media Theater." In: *Proceedings of the 14th International Conference on Entertainment Computing (ICEC-2015).* Springer, 2015

[KSG15a] Marcel Köster et al. "An Interactive Planetary System for High-Resolution Media Facades." In: *Proceedings of the International Symposium on Pervasive Displays. International Symposium on Pervasive Displays (PerDis-15), June 10-12, Saarbrücken, Germany.* ACM, 2015

Demo, talk and application papers:

[KGK19b] Marcel Köster et al. *Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction.* 12th International Symposium on High-Level Parallel Programming and Applications (HLPP-2019). 2019

[KSG15b] Marcel Köster et al. "Gravity Games - A Framework for Interactive Space Physics on Media Facades." In: *Proceedings of the International Symposium on Pervasive Displays.* ACM, 2015

In addition to this list, this thesis also refers to several publications that can be used to extend or adapt the methods presented. They had been created in collaboration with other researches but *do not* contribute to this thesis:

[GKK20]⊛ Julian Groß et al. "CLAWS : Computational Load Balancing for Accelerated Neighbor Processing on GPUs using Warp Scheduling." In: *Proceedings of the Conference on Computer Graphics & Visual Computing (CGCV-2020)*. The Eurographics Association, 2020

[GKK19] Julian Groß et al. "Fast and Efficient Nearest Neighbor Search for Particle Simulations." In: *Proceedings of the Conference on Computer Graphics & Visual Computing (CGCV-2019)*. The Eurographics Association, 2019

[LKH15] Roland Leißa et al. "A Graph-Based Higher-Order Intermediate Representation." In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. ACM, 2015

[Kös+14a] Marcel Köster et al. "Code Refinement of Stencil Codes." In: *Parallel Processing Letters (PPL)* (2014)

[Kös+14c] Marcel Köster et al. "Platform-Specific Optimization and Mapping of Stencil Codes through Refinement." In: *Proceedings of the 1st International Workshop on High-Performance Stencil Computations (HiStencils-2014)*. 2014

[Mem+14] Richard Membarth et al. "Target-Specific Refinement of Multigrid Codes." In: *Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC-2014)*. IEEE, 2014

[Dan+14] Piotr Danilewski et al. "Specialization through Dynamic Staging." In: *Proceedings of the 13th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 2014

CONTENTS

CHAPTER 1

GENERAL INTRODUCTION

1.1 Preface

Recently, general purpose programming on graphics processing units (programming on GPUs, referred to as *GPGPU*) has become increasingly important [AMD19; NVI23a]. Parallelizable tasks have been ported to the GPGPU world to benefit from increasing compute performance and memory throughput. However, this trend comes at the cost of making algorithms compatible with these hardware architectures [Kös+14a; KGK19a]. Related to this is another challenging problem of inventing scalable algorithms that maximize utilization on such devices (also referred to as *accelerators*) [Kri01; NVI14; NVI23a].

Ongoing research regularly proposes new algorithms that improve performance in a variety of application domains. Especially important for this work are the domains of *simulation and optimization*, which are also often closely coupled in many use cases [Gel+12; KGK19b; KGK19c]. Although there has been a significant amount of underlying methods and concepts to improve performance in both domains on accelerators, there are still larger problems waiting to be solved. For example, simulations almost always benefit from improved precision when more elements are simulated [Mül08; KK16]. In this context, an element may refer to a single particle in 3D space moved by domain-dependent simulation logic (e.g., particle-based fluid simulations [MCG03; MM13; Ihm+14]). Growing datasets or the modeling of (more) complex physical systems automatically lead to a continuously increasing demand for computational resources [HS13; Mac+14]. This is also true for visualization-based analyses of such complex systems. Even modern analysis tools are often unable to scale appropriately to or provide real-time insights into complex input data that needs to be analyzed in real time [KK18].

Similar effects have been observed in the optimization domain [ACK13; Cam+14a]. In general, optimization problems are omnipresent in our everyday life (e.g., scheduling and routing problems [Tal09]). Moreover, they are a fundamental building block when using modern *artificial-intelligence* (*AI*) based methods [RT18; KGK19a; KGK19c]. The use of *machine learning* (*ML*) made it possible to overcome limitations in solving extremely challenging problems that could not have been solved before [Gel+12; Sil+16]. Particularly, ML based-methods are often combined with other AI approaches to explore the search space for potentially interesting solutions [Mel+11; Cam+14a; Gel+12; Sil+16]. In this context, ML is often utilized to guide the search-space exploration process to achieve more promising results [KGK19a; KGK19c]. More generally, these methods are also used to implement *search heuristics* around which other meta-optimization systems are built [Tal09; KGK19c].

In the context of projects we had worked on, we had faced similarly difficult problems (see also Chapter 12). Specific time constraints for delivering solutions added another layer of complexity to the results we were looking for. Even available state-of-the-art methods (at the time of writing) were not able to satisfy all of our demands [KK18; KGK19a; KGK20b; KGK20a]. In all cases, the runtime performance was not sufficient to use existing solutions [KK18; KGK19c].

Consequently, alternative methods and new ways to overcome the limitations of all previous approaches were needed. We contributed to the fields of parallel simulations and heuristic optimization on GPUs through several (award-winning) publications. The methods developed by the contributing author in the course of writing this thesis have significantly improved the state of the art. Our high-level and low-level algorithms achieved speedups of up to several orders of magnitude compared to our directly competing methods. Most importantly, we maintained the same accuracy as related methods and did not weakened previous guarantees (where applicable). The same is true for memory consumption. Certain contributions in this work were able to reduce memory requirements significantly, by up to several gigabytes of data that would have been required.

The remainder of this chapter is intended to give the reader a more technical view of our contributions (see Section 1.2). In addition, we present our research questions in Section 1.3 and show an outline of the thesis including a description of our three parts in Section 1.4. Chapter 2 serves as an introduction to essential GPGPU programming principles. It also provides a foundation for understanding the terminology used in this thesis when discussing and describing GPU programs and our algorithmic concepts.

1.2 Technical Introduction

This technical introduction section presents the different parts covered in this thesis, as well as general terms used throughout this work. The whole dissertation is divided into two major parts followed by an engineering contribution section that sum up to a whole: *Simulations*, *Heuristic Optimization* and *Optimizer Generation* (see Section 1.4). *Part I* covers research results from the domain of parallel simulations on GPUs. *Part II* presents contributions made to the field of (heuristic) optimization.

While the first two parts focus on the scientific and algorithmic contributions, *Part III* summarizes our engineering contribution. It outlines our compiler-based approach for generating specialized optimizers to solve simulation- and heuristics-based optimization problems using the methods presented in this dissertation. This compiler concept to automatically generate optimizers has been successfully applied to many different industry-grade optimization problems that could not be solved by traditional methods considering their time constraints.. Part III concludes the entire thesis and all parts and gives an outlook on future work.

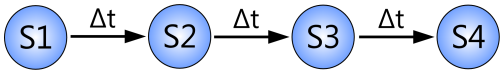

Figure 1.1: A high-level simulation workflow with three simulation steps.

Starting with simulations, Figure 1.1 visualizes an high-level abstract *simulation workflow* using different *states* S_1 to S_4 with three *simulation steps*. A state refers to a modeled and instantiated representation of the underlying simulation domain. Applying a simulation-specific function to a state S_i results in a *successor state* S_{i+1} that contains all updates performed by the *step function* (also referred to as *update function*). In this case, we start with an initial state S_1 and apply the step function three times in a row using a fixed *time-step size* Δt, where Δt represents the simulation time that elapses between two different states. Note that Δt can refer to both a physically-based and a logically-based time-step size, depending on the actual simulation problem. This also means that Δt can be represented by different data types or even complex data structures.

The presented simulation methodology can also be used in the scope of solving optimization problems. In this context, the actual problem description is represented using a "game-like" simulation logic (similar to the rules of a chess game, for instance) combined with an exploration and search-based solver approach. There is a huge variety of different possibilities how to realize such an exploratory search. We focus on using the concept of *search trees* in terms of steps, expansion, pruning and traversal algorithms. However, the work presented in this thesis divides the notion of a search tree into two distinct high-level phases: *simulation logic* and *successor generation*. Although the

ideas of pruning and traversal are not explicitly addressed in this thesis, all work presented here can be combined with any existing approach from these research fields. Figure 1.2 visualizes our high-level search-tree concept, which is consistent with all contributions in this thesis.

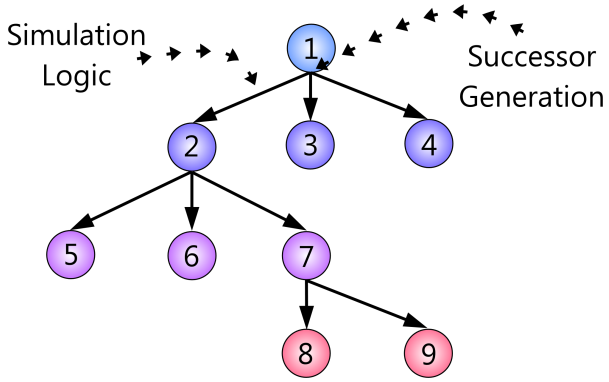

Figure 1.2: A high-level search tree of a simulation-based optimizer instance [KGK19c].

As before, we start with an initial state S_1 (denoted as 1 in the diagram for simplicity). During *successor generation*, possibly interesting successor states will be created $(2, 3, 4)$. We then apply simulation steps an *evaluation function* (also referred to as *cost function*) to each state. This is necessary to select the potentially interesting/best states that are believed to be most likely to lead to an optimal solution of the problem, taking all constraints into account. After selecting several states and progressing a certain number of steps in this phase, we again reach the *successor generation* phase.

All contributions in chapters throughout Part I and Part II serve as the foundation for building a fully functional GPU-accelerated optimizer leveraging heuristics. To achieve high performance on these accelerators, it is essential to choose appropriate memory layouts for all involved *memory buffers* (e.g., contiguous arrays allocated on a specific device) and to use specialized versions of the presented algorithms in the first two parts of this thesis (see also Chapter 2). Additionally, various target platforms provide specific low-level features to further improve performance.

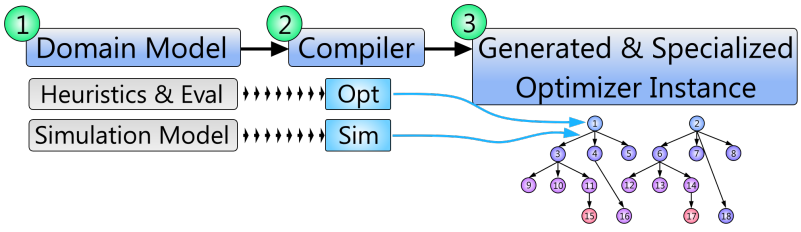

Figure 1.3: Our high-level workflow to generate optimizers.

Figure 1.3 illustrates our compiler-based approach in this work to generate specialized optimizers in Part III, representing our engineering contributions. Our approach relies on a problem-specific *domain model*, which contains essential simulation logic, evaluation functions, and heuristics to guide tree-based search (1, *left*). The task of the compiler in the second step is to transform the previously created domain model into a task- and problem-specific optimizer (2, *middle*). By leveraging this domain model, we can efficiently generate task- and problem-specific optimizers in the second step. The resulting emitted optimizer (3, *right*) can then be launched alongside a problem-specific *problem instance*, ensuring compatibility between the input domain model and the generated optimizer.

1.3 General Research Questions

A single central meta-research question RQ_M was the main driving force for all contributions and this thesis. It was derived from the continuously advancing trend to improve performance and is as follows:

RQ_M Given next-generation massively parallel hardware accelerators, how can we improve runtime performance of computationally challenging tasks?

As mentioned (and briefly discussed) in the abstract and in both the general and technical introductions, modern architectures require adaptation (or even rethinking) of the basic algorithms used. The general research direction for this work focused on novel algorithms and concepts in favor of small and negligible contributions. Therefore, we decided to split the main question into several fine-grained sub-questions that describe each topic in more detail:

RQ_1 How to improve performance of domain-specific and generic parallel simulations by building on current GPU characteristics?

RQ_2 How to improve solving performance of optimization problems by building on current GPU characteristics and leveraging the improvements of parallel simulations from RQ_1?

In this context, the term performance refers to either runtime performance or performance in the sense of improved memory consumption. We primarily focus on runtime performance improvements compared to state-of-the-art approaches in most contributions. However, we have also addressed several significant memory-consumption issues blocking algorithms from scaling appropriately to larger problems.

Although answering these general research questions is still subject to ongoing research, we focus on contributing to answering these guiding research questions that formed the basis for our research. The various parts of this dissertation (see Section 1.4) aim to answer related questions one and two.

1.4 Outline of the Thesis

As mentioned in the technical introduction, the three parts of the thesis focus on different topics and contributions. Figure 1.4 shows all three parts and their individual contribution to our guiding research questions. In the scope of this thesis, Parts I and II have a similar structure. After introducing each topic, we present each individual contribution to the associated field of research. Every paper contributed to each part is explicitly presented, distinguishing between own and co-authored contributions. We then discuss related work, conceptual differences, and limitations. Each part has several main chapters introducing the actual concepts and new algorithms introduced in this thesis.

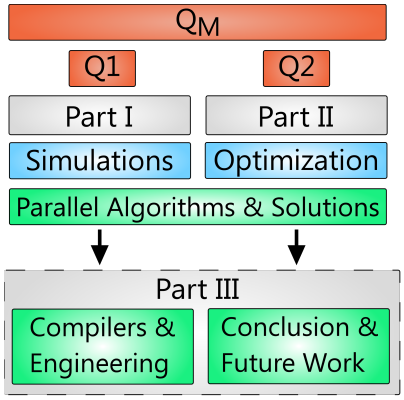

Figure 1.4: Outline of the thesis with all three parts and their contribution to answering our research questions.

Parts I and II cover parallel algorithms and solutions, while Part III covers our main engineering contributions that build on the algorithms in this thesis. Part III also summarizes all contributions and previews future work.

CHAPTER 2

GPU PROGRAMMING FUNDAMENTALS

This chapter focusses on introducing key concepts and terms that are essential for the algorithms and concepts in this thesis. The primary objective is to familiarize readers with fundamental architectural aspects unique to GPUs we were targeting at the time of publication of this work. The interested reader may refer to [AMD19], [NVI23a], and [Kös23] for more information about this topic in general.

2.1 Basic GPU Architecture

This section gives a high-level introduction into GPU architectures using NVIDIA's Ampere architecture as the basis [NVI23a; NVI23b]. At the time of writing, this GPU architecture was a state-of-the-art architecture for GPGPU.

Figure 2.1 shows an image of a high-level GPU architecture while focusing on GPGPU capabilities. Taking the general NVIDIA GPU design into account, an accelerator consists of multiple *Graphics Processing Clusters (GPCs)* which are connected to memory controllers and in turn to main GPU memory. Each GPC consists of several *Streaming Multiprocessors (SMs)* [NVI23a; Kös23] [1] and has access to a shared L2 cache, which is mainly automatically managed. Most recent devices have the ability to manage *cache residency*, giving developers control over whether the cache contents should be invalidated between program runs or kept partially [NVI23a].

[1] A GPC also contains computer-graphics specific units, which are are not shown here. For instance, *Raster Engines* to perform accelerated rasterization of geometry [NVI23a]. Note that on actual hardware, there is also another hierarchy level between GPCs and SMs called *Texture Processing Clusters (TPCs)* featuring specific texture acceleration circuits. However, since our focus lies on giving an introduction into general purpose computing on accelerators, we elide this information for the sake of simplicity and focus on SMs.

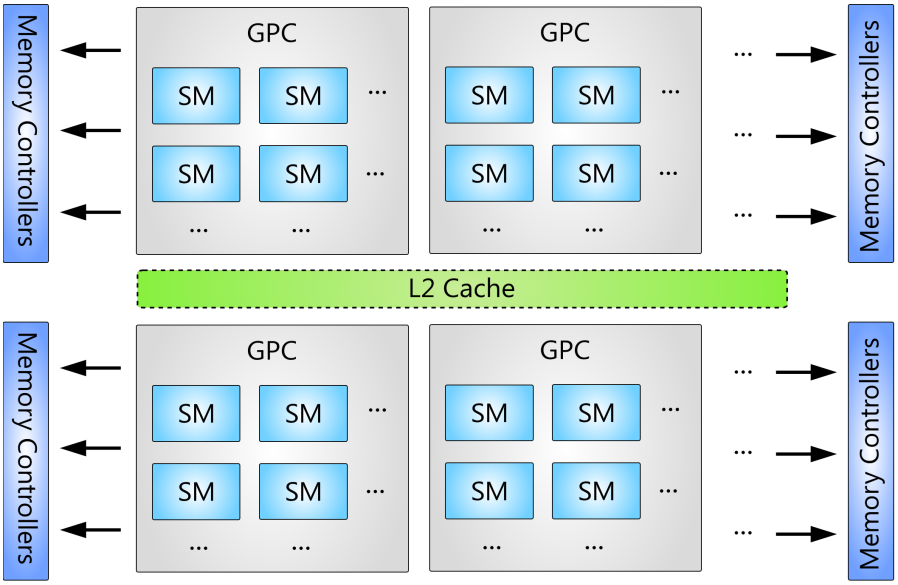

Figure 2.1: High-level GPU architecture while focussing on GPGPU capabilities based on the NVIDIA Ampere architecture [NVI23a; NVI23b]. A set of *Graphics Processing Clusters* (GPCs) consist of nested *Streaming Multiprocessors* (SMs). Each GPC is connected to shared L2 cache while also having connections to memory controllers in order to access GPU main memory. Note that the number of actual GPCs on a particular device is architecture, revision, and even vendor specific.

Figure 2.2 visually represents the core components of SMs, highlighting that all SMs share the same internal structure in terms of their underlying hardware capabilities. They are primarily made of *Stream Processors (SPs)*, which are responsible for performing most of the computational tasks on a GPU, as they contain the actual compute units and register banks. Moreover, SPs also contain separate instruction caches, schedulers to distribute workload within a SP, and compute units (ALUs, FPUs, and specific tensor operation acceleration units, TPUs) [2]. However, SMs are equipped with dedicated L1 cache that can also be used programmatically to store intermediate values explicitly in fast on-chip memory [NVI23a; Kös23] (see also Section 2.2). The primary source of computational power originates from the vast number of compute units per SP, multiple SPs per SM, and in turn multiple SMs per GPC. This sums up to an enormous amount of compute units and memory controllers, as well as large amounts of L1 cache that can be leveraged using massively parallel programs. Note that the total number of SPs and SMs on a specific device is architecture, revision, and even vendor specific.

[2] Individual ALU, FPU and TPU units are not visualized explicitly as they are considered to be subsumed by the term *compute units* for the sake of simplicity.

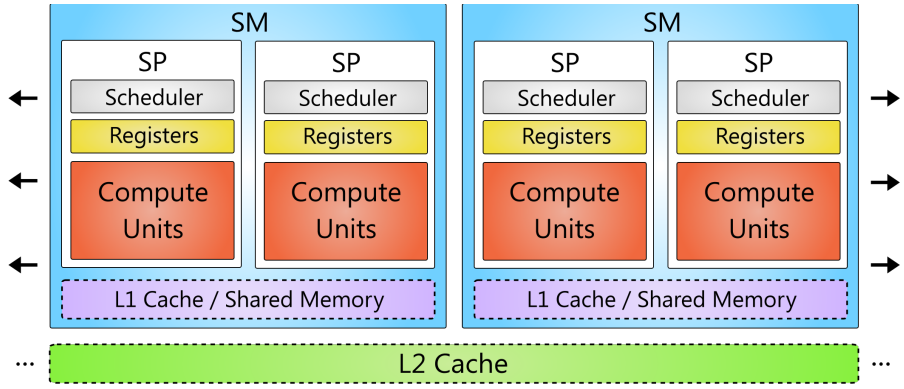

Figure 2.2: High-level architecture of *Streaming Multiprocessors* (SMs) while focussing on GPGPU capabilities based on the NVIDIA Ampere architecture [NVI23a; NVI23b]. This visualization does not contain surrounding GPCs, as the SMs can be conceptually seen as being connected to L2 cache while programming. Moreover, this is actually the case on other GPUs from different vendors, where the actual SMs are not logically grouped into GPCs. Referring to NVIDIA's architectures, a SM can contain multiple *Streaming Processors* (SPs) while L1 cache is shared across all SPs in an SM. Note that the names GPC and SM are also NVIDIA specific. For instance, AMD uses the more general term *Compute Unit* to refer to grouped processing units (similar to SMs) for their devices [AMD19].

To write parallelized programs targeting accelerators, a common way is to program a single thread of execution that can be run many times concurrently to gain advantage of the available compute and memory units [Kös+14a; NVI23a]. The basic idea is to distribute the work among all processing units to achieve maximum utilization with respect to the available processing capabilities [Amd67; Kri01; Kös+14c; Kös+14b; NVI23a]. Such a program in the scope of an accelerator is referred to as a *kernel* in the scope of this thesis.

Besides parallelization across all SMs (and ultimately all SPs), Streaming Processors leverage the concept of *vectorization* on the compute and even on the memory-controller level [Kös+14b; Dan+14; NVI14]. From an abstract point of view, vectorization is based on the fundamental idea of *SIMD* (single instruction multiple data). It allows the same operation to be performed on multiple data elements simultaneously. To be precise, the concept used in GPUs is *SIMT* (single instruction, multiple threads) [TCS20; NVI23a]. It combines the SIMD model with multithreading by utilizing multiple data elements of the SIMD model to represent multiple threads running in parallel. We refer to single SIMT unit by a *warp*, which is usually consists of 32 threads on NVIDIA GPUs and 64 or 32 threads on AMD GPUs depending on the architecture.

An important detail to be aware of is the fact that using SIMT means all instructions in the same warp will execute in lockstep [LSG19; AMD19; NVI23a]. Lockstep in this scope means that the same operation will be executed on all processing threads inside a warp. This implies that if a program has divergent control flow (e.g., by using an if-statement with side effects in the input program), this will cause the control-flow to be serialized and all statements of both branches will be executed one after another (see Figure 2.3).

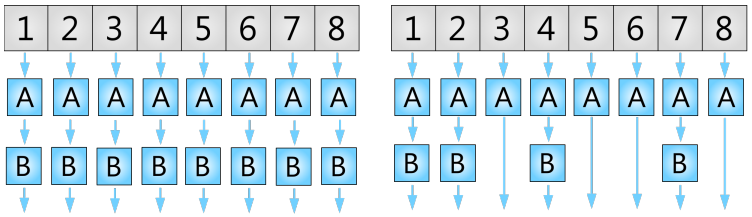

Figure 2.3: Lockstep execution semantics of a program with two blocks A and B [KGK20b] on an imaginary *warp* with eight threads (left). Divergent control flow leads to a non-optimal occupancy of a warp by executing block B in several threads, while the others have to "wait" (idle) until all threads can continue with the next block (right).

2.2 General Terminology

We follow the naming scheme of the ILGPU framework [Kös23] (see Section 2.3). In this context, we refer to a processing thread on a GPU as *thread*. When multiple threads are dispatched in the scope of a kernel launch, they may run concurrently with respect to each other, as mentioned before. Within a warp, we call a thread a *lane*. The position of a lane within a warp is referred to by its *lane index*. As the warp size depends on the actual hardware (e.g., 32 for NVIDIA and 64 or 32 for AMD), the size of a warp on a GPU in our algorithms is referred to by *warp size*.

Taking GPU architectures into account, a set of threads can be *grouped* logically to share information. These groups can also span multiple warps up to a hardware and vendor-specific maximum size. The number of threads per group is denoted by the *group size*. Currently available GPUs typically have a smaller (or equal) maximum group size compared to the maximum number of concurrent threads per multiprocessor. This ensures that groups are always executable on a single multiprocessor, as described in detail in [NVI23a; Kös23]. Within each group, threads are identified based on their *group index*, which represents the current thread's relative position within that group. The position of a warp within the current group is referred to by *warp index*.

As a kernel launch can consist of multiple groups being dispatched at the same time, all groups executed in a *thread grid* (or brief *grid*) [NVI23a; Kös23]. The actual number of groups being dispatched is denoted by the *grid size*. Although certain APIs expose multidimensional groups and grids to be dispatched, we realize GPGPU operations in 1D thread grids. This also enables use of our methods and algorithms on a huge variety of target platforms and APIs.

Memory Types In the scope of this thesis we follow NVIDIA's classification of memory types and follow their naming scheme [NVI23a; Kös23]. Therefore, we differentiate between the following memory types:

- Constant memory
  is a small amount of read-only memory (a few KB) that is cached for fast access times inside a kernel. It can be used to pass information from the *host* system (the CPU-based part of the application) to a kernel. Depending on the architecture and the vendor of the device, only a subset of the totally available constant memory may be cached upon launching a kernel [NVI23a].

- Shared memory
  is a small, highly efficient and very fast on-chip region of memory that can be used to store intermediate results and share information between different threads within a thread group. The amount of shared memory varies depending on the architecture and the vendor of the device. Recent NVIDIA GPUs (at the time of writing this thesis) offer up to 164KB shared memory per multiprocessor [NVI23a]. Please note that the amount of shared memory being used has an implication of the maximum number of groups assignable to a multiprocessor [NVI23a; Kös23]. This kind of memory is also referred to as *scratchpad* or *local* memory in the context of specific APIs and vendor specifications [AMD19]. Furthermore, shared memory is considered to be group-specific in terms of that only threads belonging to the same thread group can share information between each other through shared memory.

- Local memory
  is called local as it is thread local or private to each processing thread. Despite its name, it resides in global GPU memory and is the default target location for large local arrays, indexed data structures that cannot be operated on in register space, and register spilling. Lastly named occurs when the number of available registers is exceeded [AMD19; NVI23a; Kös23]. On modern NVIDIA GPUs, local memory is cached in L2 cache [NVI23a].

- Global memory
  refers to main GPU memory which can contain up to several GB of data at the time of writing this thesis [AMD19; NVI23a]. Its access latency is considerably higher than the latency of shared memory and accesses to this type of memory should follow specific access patterns for best performance (see Section 2.4).

2.3 Programmability and APIs

There is a huge variety of different APIs and programming languages available to target GPUs in general. Shading languages for computer graphics (and even compute) applications, domain-specific languages to model specific use cases [Kös+14a; Mem+14], and general purpose GPGPU languages/language extensions [NVI23a; Kös23].

As our contributions focus on novel algorithms in the GPGPU space while optimizing for overall runtime performance and scalability, we chose general-purse GPU programming environments for our tasks. We mainly realized our GPGPU workloads in either NVIDIA's CUDA [NVI23a] combined with C++ and C# while using ILGPU [Kös23] for all GPU kernels. We did not leverage OpenCL [Gro23] as a kernel language directly due to practical limitations and often reduced performance on our primarily targeted NVIDIA GPUs.

CUDA was primarily chosen for specific conferences to have comprehensive measurements in comparison to other related papers published in certain domains. In all other cases in which this was not required, we realized all GPU kernels and the surrounding host applications in C#. This also included reference implementations we reimplemented from related work for comparison purposes.

The ILGPU compiler and runtime framework was chosen as it was a state-of-the-art approach to realize truly portable GPGPU programs. Portable in this context means that we only had to compile our program once using standard .Net toolchains and could deploy it on arbitrary machines that supported .Net and had GPU driver installed. ILGPU acted as just-in-time compiler sitting on top of the .Net runtime compiling our code to either PTX or OpenCL-compatible representations. As it also supported multithreaded CPU code generation, there was always an efficient fallback mode available. This made ILGPU a perfect tool and environment to realize our implementations and deploy them seamlessly on arbitrary target systems without recompiling.

ILGPU ILGPU itself was started as a hobbyist project by the contributing author to provide a unified GPGPU experience in the .Net world. Over time, it has become a popular and enterprise grade compiler written entirely using .Net languages to be also truly portable to all .Net compatible environments. Besides extremely convenient debugging capabilities using its build-in GPU emulator, it offers en-par performance with the CUDA toolchain or other LLVM-based compilers for GPU computing [Kös23].

General Implementation Details All contributions, in particular all algorithms, do not use local memory explicitly. They are designed in a way that all intermediate processing data can reside in registers and shared memory for performance reasons. Moreover, we gave additional hints in our implementations to GPU drivers by passing information about:

- the maximum number of threads per group, and
- the minimum number of groups per multiprocessors.

This helped GPU driver backends to adjust register allocation accordingly to match our use cases.

Synchronization and Atomic Operations In the scope of this thesis, we make use of group-wide synchronization primitives and atomic operations in shared and global memory. Synchronization primitives we use are limited to *group barriers* that act as well known thread barriers for all threads in a group. We assume that these barriers also work as explicit group-wide memory fences that ensure each thread in the group can see all memory operations from other threads in the group after all threads have passed the barrier.

Atomic operations give the ability to perform lock-free operations while ensuring that multiple threads access data in a consistent manner (referred to as *atomics*). This prevents race conditions and other synchronization issues while avoiding locking in all cases [Kös23]. Note that the concept of atomics is also available on CPUs via specific instructions. We make use of atomic operations on 32-bit and 64-bit integers and 32-bit floats in shared and global memory. For performance reasons, we still consider these operations as being accelerated in hardware on the target devices. In case of missing hardware capabilities to perform atomic operations on floats, it is also possible to emulate them in software [Kös23].

The interested reader may refer to [NVI23a; Kös23] for detailed information about limitations in the context of synchronization and thread-safe resource accesses.

2.4 Memory Accesses

As mentioned before, global memory is much slower than shared memory in terms of its latency and throughput. Accesses to this kind of memory should follow specific *coalesced memory-access patterns* to achieve maximum performance [AMD19; NVI23a; Kös23]. Coalesced in this context refers to the property that a thread access "neighboring addresses" in global memory with respect to neighboring threads (see Figure 2.4).

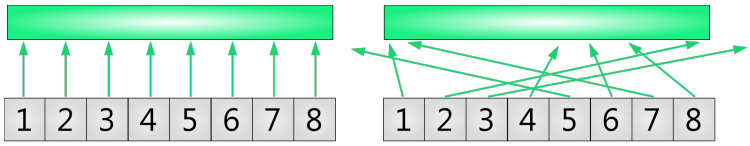

Figure 2.4: Schematic visualization of eight threads in warp performing memory accesses. Left: Coalesced global memory accesses to neighboring cells. Right: Accesses to arbitrary memory locations.

As shown in the diagram (right), random accesses to arbitrary memory addresses from different threads violate the coalesced memory access pattern. This can in turn lead to considerably reduced performance [NVI23a]. There are exceptions to this general rule of thumb in which the memory addresses are permuted across multiple threads and their general access window is limited to a certain distance between memory addresses being accessed [NVI23a; Kös23]. Depending on the actual GPU hardware being used, coalesced memory accesses can be important on the warp and/or the group level [KGK19a; Kös23]. To make our contributions in this thesis as general as possible, we are not relying on the assumption of whether coalescing should be applied on warp or group levels. Instead, all algorithms make excessive use of coalesced memory accesses on the group and warp level whenever possible.

An example in which we are relying on non-coalesced memory accesses is using reductions to compute group-wide or device-wide information. Reductions on the warp-level do not require memory accesses to global and shared memory in our domain.

Combining this information with SIMT-based control flow, makes things more challenging to master. Figure 2.5 shows the influence of complex control on memory access patterns while also taking the overall warp utilization into account (see also Section 3.1 for a real-world example). Consider a kernel which consists of nested if statements, as shown in Algorithm 1. For the sake of simplicity, this simple sample program is meant to be run in a single group (*grid size = 1*) and takes four output buffers which are written to by the kernel. Note that we assume all buffers do not alias and do not overlap. Each thread can write to all of four memory buffers depending on the evaluation of the if conditions inside the kernel. The case differentiations cause memory accesses to be non-coalesced and do not achieve maximum global-memory performance.

Algorithm 1: Memory accesses in the presence of complex control flow

```
   Input: Output buffers output1, output2, output3, and output4
   /* Block A                                                    */
1  output1[group index] := ...;
2  if ... then
   |  /* Block C                                                 */
3  |  output2[group index] := ...;
4  |  if ... then
   |  |  /* Block D                                              */
5  |  |  output3[group index] := ...;
6  |  |  if ... then
   |  |  |  /* Block E                                           */
7  |  |  |  output4[group index] := ...;
8  |  |  end
9  |  end
10 end
11 if ... then
   |  /* Block B                                                 */
   |  /* ...                                                     */
12 end
   /* ...                                                        */
```

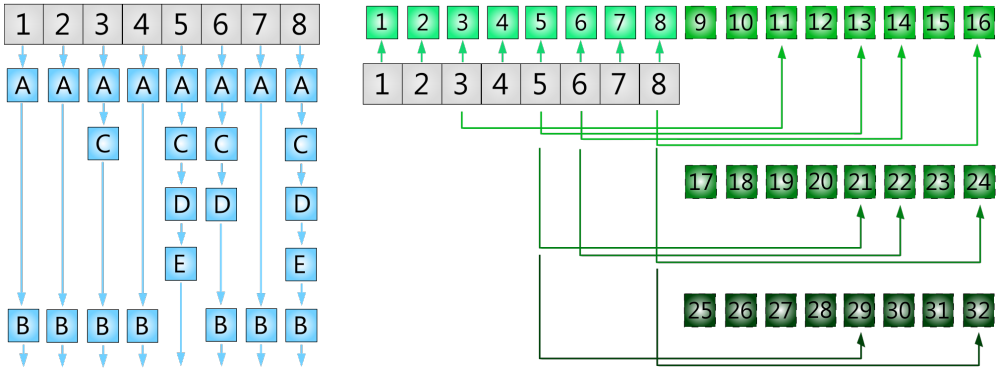

Figure 2.5: Thread divergence within a warp (gray) caused by complex control-flow logic consisting of multiple blocks from Algorithm 1 (A–E, left) [KGK20a]. Influence of control-flow divergence on the memory accesses to individual elements in global memory in the scope of all blocks (right). Different output buffers are denoted by different shades of green. Light-green arrows highlight schematic accesses in block A (elements 1–8 in this sample, right), whereas dark-green accesses (elements 25–32 in this sample, right) indicate memory accesses in block E. Accesses in blocks B (elements 9–16, right) and C (elements 17–24) are highlighted with slightly darker green arrows compared to accesses in block A.

Figure 2.5 visualizes memory access patterns of Algorithm 1 when launching a single warp only on an imaginary GPU with a warp size of eight. Moreover, the visualization assumes a single chunk of allocated GPU memory that can be accessed contiguously (memory addresses 1 to 32 in the diagram). This sample demonstrates that control-flow divergence does not only influence the utilization of warp lanes in terms of compute, but also the way previously designed coalesced memory accesses may not be issued in the optimal way regarding memory throughput. Although Algorithm 1 does not represent a real-world example, it clearly presents the complexity involved when optimizing algorithms for overall utilization in terms of compute and memory efficiency.

Shared Memory Accesses Accesses to shared memory should also follow certain characteristics to achieve best performance. Focussing on NVIDIA GPUs, coalesced memory accesses to shared memory can be beneficial. However, very important performance wise is to avoid *bank conflicts* when performing shared-memory operations.

Shared memory on NVIDIA GPUs is separated into *banks* of either 32-bit or 64-bit depending on the available hardware capabilities that can be programmatically defined [NVI23a; Kös23]. In case of multiple threads writing to the same address in shared memory, a bank conflict may occur. This results in serialized accesses, which in turn cause performance degradation.

Compute and Memory Bound Kernels The term *compute bound* in GPGPU refers to kernels that primarily depend on large amounts of compute. For these kernels, the available computational resources are the main bottleneck. *Memory-bound* kernels are affected by the overall memory performance, which is the bottleneck in these cases. While a strict categorization of whether a kernel is compute or memory bound is often possible, many programs show characteristics of both kinds. In turn, they benefit from compute focused, as well as, memory focused optimizations.

Nevertheless, using shared memory is often extremely beneficial for memory-bound problems to improve performance considerably. For instance, prefetching chunks of global memory for further processing is a common pattern. It is often used to buffer non-coalesced reads from global memory or to cache intermediate results in terms of compute-driven kernels.

Nearly all of our problems we target in this work are memory bound. Therefore, we make excessive use of shared memory. This allows us to overcome global-memory performance limitations, which leads to significantly improved performance of our algorithms. Reconsidering compute-bound problems in our space, we also use shared memory to cache intermediate results and propagate them to all threads in the same group. This way, we can avoid expensive recomputations and minimize (even avoid in most cases) transfers of intermediate results to global memory in general.

Part I

Parallel Simulations

CHAPTER 3

INTRODUCTION

Simulations are commonly used in a variety of application domains. In particular, large-scale simulations have become more and more popular to investigate specially challenging problems, e.g., in the fields of physics and applied mathematics. Many simulations in these domains operate on elementary data elements, often referred to as *particles* in the case of 3D problem definitions. To get a better understanding of the structure of such tasks, we consider the N-Body problem from physics [KSG15b; Ngu07][Chapter 31] as a running example that is well known and has been studied for a long time [BH86]. From a high level point of view, it generally describes the problem to compute the positions of a set of *objects* in 3D space interacting with each other via their gravitational forces. Figure 3.1 shows a visualized N-Body gravity simulation on a sample dataset.

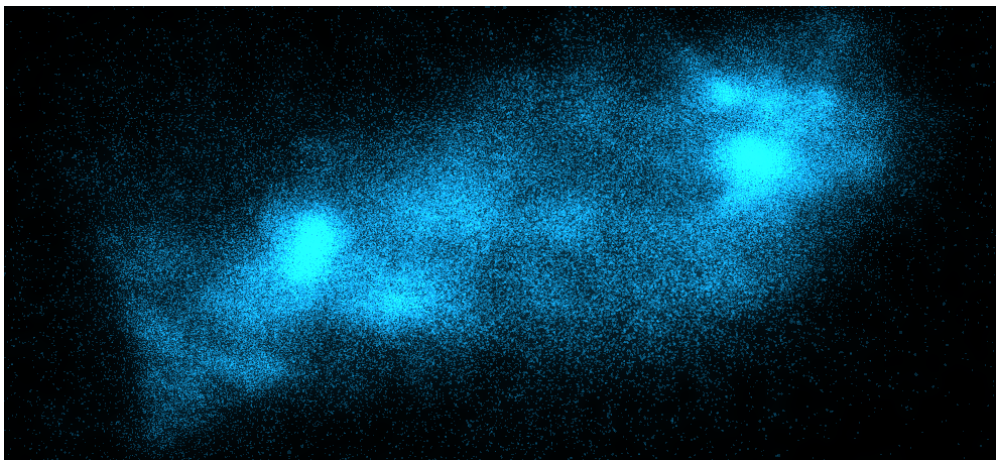

Figure 3.1: Sample N-Body gravity simulation rendered using the implementation described in Section 5.2.

The actual simulation operates on a set of *bodies* (also referred to as *masses*) represented by their *position* and *v*elocity vectors in 3D and finally their *mass* (usually defined in kilograms):

$$\text{body}_i = (\vec{p}, \vec{v}, m), \tag{3.1}$$

where $\vec{p} \in \mathbb{R}^3, \vec{v} \in \mathbb{R}^3, m \in \mathbb{R}$ and $i \in [0, \ldots, N-1]$. Here, i denotes the ith body in the scope of the simulation and N the number of bodies. [1] The actual problem is based on the force computation between two bodies based on Newton's law of gravity. The following equation defines the vector $\vec{f}_{ij}$ representing the force that the jth body applies to the ith body [KSG15b; Ngu07][Chapter 31]:

$$\vec{f}_{ij} = G\frac{m_i m_j}{||\vec{d}_{ij}||^2}\hat{d}_{ij}, \tag{3.2}$$

where G is the gravitational constant, $\vec{d}_{ij}$ is the distance vector $(\vec{p}_j - \vec{p}_i)$ between the two bodies and $\hat{d}_{ij}$ is the normalized distance vector. Based on Equation (3.2), we can compute the total force vector $\vec{F}_i$ that is applied to a single body via

$$\vec{F}_i = Gm_i \cdot \sum_{j \in \{0,\ldots N-1\} \setminus \{i\}} \frac{m_j d_{ij}}{||d_{ij}||^3}. \tag{3.3}$$

A straight forward way to implement a simulation based on Equation (3.3) is to use a nested loop that computes all forces between all pairs of objects (see Algorithm 2).

Algorithm 2: Sequential N^2 algorithm to perform a single step of an N-Body gravity simulation based on Equation (3.3)

```
   Input: N bodies, time step size Δt
   /* Iterate over all bodies                                   */
1  for i := 0 to N − 1 do
   |  /* Compute the total force vector F                       */
2  |  F_i := G · m_i;
3  |  for j := 0 to N − 1 do
4  |  |  if i ≠ j then
5  |  |  |  F_i := F_i + (m_j d_ij) / ||d_ij||^3;
6  |  |  end
7  |  end
   |  /* Update velocity vector                                 */
8  |  v_i = v_i + F_i · Δt;
   |  /* Update position                                        */
9  |  p_i = p_i + v_i · Δt;
10 end
```

[1] We use $\vec{p}_i$, $\vec{v}_i$ and m_i as abbreviations to refer to the individual properties of body_i.

In this version, we iterate over all bodies and accumulate the total force vector $\vec{F}$ applied to each body by iterating over all other bodies in the dataset. Note that this simplified pseudo code does not account for collision checks, which could be used to avoid extremely small denominators. After computing the total force, we update the velocity and position of each particle accordingly. Here, Δt represents the time that elapses between two simulation steps and is referred to as the *time-step size*. It describes a fundamental property of a simulation, which mush be chosen with a focus on precision and runtime performance depending on the individual requirements.

3.1 Parallel Simulations

As outlined in Algorithm 2, a simply version to implement an N-Body gravity simulation is to use nested loops. Each body is processed sequentially. This also means that the position and velocity of each body is automatically updated in place. In other words, the next body to be processed already sees the updated position of the previous body and takes these changes into account. At a high level, all bodies are conceptually updated at the same time (in parallel), using their original positions (source positions at the beginning of each step).

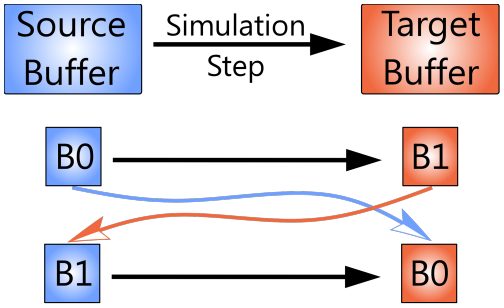

Figure 3.2: A double-buffering approach (top). The data is read from the source buffer during a simulation step. The target buffer contains the updated information after completing the step (top right, *b1* center). Afterwards, both buffers will be swapped and the original source buffer *b0* becomes the new target buffer (bottom).

Double buffering can overcome this limitation by using a *source buffer* (read only) and a *target buffer* (write enabled). The *source buffer* contains all information that was "valid" at the beginning of the time step, whereas the *target buffer* will be updated during the simulation step [2] (see Figure 3.2). After completing a single time step, both buffers will be swapped and the previous target buffer becomes the new source buffer.

[2] Note that it is not possible to rely on the contents/the integrity of the data stored in the target buffer while performing a simulation step. If it is necessary to read data from the target buffer, developers need to be extremely careful to avoid race conditions.

Reconsider our N-Body gravity example. When porting our simple sequential algorithm from the beginning (see Algorithm 2) to a parallel target device, we cannot control the update order of all bodies. This leads to nondeterministic simulation behavior when the updates are performed in place, without paying attention to possible data races that might occur and even lead to invalid simulation data. Using the presented concept of *double buffering* circumvents these problems, as all parallel threads can see the same input at the same time. One possible parallelization model for this simulation is to process each body in parallel in its own thread. This implies that we do not need to program two explicit loops any more, since each thread only needs to iterate over all other bodies.

Algorithm 3: Simple Parallel N^2 algorithm for one thread to perform a single step of an N-Body gravity simulation based on Algorithm 2. Note that each thread processes exactly ony body.

Input: N body input buffer, N body output buffer, i current body index, time step size Δt

```
/* The current body index i is already an input          */
/* Compute the total force vector F by iterating over all
   other bodies                                          */
```

1 $\vec{F_i} := \vec{0}$;
2 **for** $j := 0$ **to** $N - 1$ **do**
3 | **if** $i \neq j$ **then**
| | /* Compute force based on the input buffer */
4 | | $\vec{F_i} := \vec{F_i} + \frac{m_j \text{in}(d_{ij})}{||\text{in}(d_{ij})||^3}$;
5 | **end**
6 **end**
/* Compute updated velocity vector $\vec{v}$ */
7 $\vec{v} = \text{in}(\vec{v_i}) + \frac{\vec{F_i}}{m_i} \cdot \Delta t$;
/* Update velocity vector in output buffer */
8 $\text{out}(\vec{v_i}) = \vec{v}$;
/* Update position in output buffer */
9 $\text{out}(\vec{p_i}) = \text{in}(\vec{p_i}) + \vec{v} \cdot \Delta t$;

3.2 Improving Performance

Regarding runtime performance, the inner loop (lines 2 to 6 in Algorithm 3) is the performance critical part of the simulation. This loop involves memory accesses to the positions of all other bodies in the simulation and an if condition that often leads to suboptimal occupancy of warp lanes (see Chapter 2). Moreover, different arithmetic operations take more time (more processing cycles) to complete. A common example is floating point division, which is often significantly slower than multiplying two values [NVI23a]. Such optimizations can be left to the compiler to do performance-aware transformations.

To improve performance without relying on program-transformation techniques, it may be possible to restructure the underlying simulation specification (i.e., the underlying equations). The goal would be to redesign the problem description in a way that avoids expensive operations and corner cases. In the case of the running example, we can rewrite the denominator of the force computation to avoid cases where bodies (really close to each other) get very large forces, as suggested in [Ngu07][Chapter 31]:

$$\vec{F_i} = Gm_i \cdot \sum_{j \in \{0,\dots N-1\} \setminus \{i\}} \frac{m_j \hat{d_{ij}}}{||d_{ij}||^2} \tag{3.4}$$

$$= Gm_i \cdot \sum_{j \in \{0,\dots N-1\} \setminus \{i\}} \frac{m_j d_{ij}}{||d_{ij}||^3} \tag{3.5}$$

$$\approx Gm_i \cdot \sum_{0 \leq j < N} \frac{m_j d_{ij}}{(||d_{ij}||^2 + \epsilon^2)^{\frac{3}{2}}}. \tag{3.6}$$

By adding an additional small ϵ softening value allows us to avoid extremely large forces and makes the innermost *if condition* $i \neq j$ (lines 3 to 5) unnecessary. Depending on the use case, using an approximate reciprocal function $\text{rcp}(x) = \frac{1}{x}$ instead of an expensive floating-point division helps to improve performance (at the cost of simulation precision [Ngu07; Kös13]). The same method can also be used for the function $\sqrt{x}$, which is usually available in the form different approximations with different precisions [Ngu07]. The following code snippet demonstrates the use of approximation methods to implement Equation (3.6):

```
1  distanceSquared = dot(r_ij, r_ij);
2  denominator = rsqrtApprox(distanceSquared + epsilonSquared);
3  denominator = denominator * denominator * denominator;
```

Regarding memory accesses, the access pattern in our running example is coherent, as all threads load the first body first, then the second, and so on (see Chapter 2). However, loading bodies from global memory is not optimal, as multiple threads can load different bodies at the same time while accessing neighboring bodies in memory. We can also drastically reduce the number of loads by leveraging a small body cache in shared memory (see Figure 3.3).

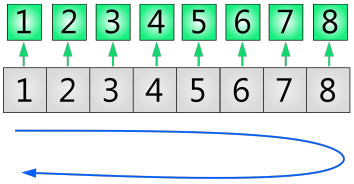

Figure 3.3: Conceptual functionality of a shared-memory cache based on the sample of a gravity simulation. First, all neighboring threads load their associated bodies into shared memory (green arrows). Afterwards, all threads perform the force computation across all bodies in the group using the data stored in shared memory (blue arrow). Finally, all threads continue with the next bodies.

Algorithm 4: Conceptual parallel N^2 algorithm using shared memory based on Algorithm 3 [Ngu07; Kös13]

Input: N body input buffer, N body output buffer, i current body index, time step size Δt

```
   /* Allocate shared memory for each body                          */
1  sharedBodies := shared memory (float3, float)[group size];
   /* Compute the total force vector F by iterating over all
      other bodies                                                  */
2  F_i := 0;
   /* Iterate over all bodies with a stride of the current
      group size                                                    */
3  for j := 0 to N − 1 step by group size do
   |  /* Load a body into shared memory using coherent
   |     memory accesses                                           */
4  |  sharedBodies[group index] := in(j + group index);
   |  /* Wait for all threads to complete the load operation
   |     into shared memory                                        */
5  |  group barrier;
   |  /* Iterate over all cached bodies                           */
6  |  for k := 0 to group size − 1 do
   |  |  /* Update the current force vector F using
   |  |     sharedBodies here                                     */
7  |  end
   |  /* Wait for all threads to avoid overwrites from
   |     thread that are still iterating over the
   |     shared-memory contents                                    */
8  |  group barrier;
9  end
   /* Update velocity and position...                              */
```

Leveraging shared memory can be achieved by blocking the loop that iterates over all other bodies. Blocking means that we adjust the step size of the outermost loop to be incremented by the current *group size* (the number of threads in each group). Algorithm 4 shows a sample implementation using shared memory based on Algorithm 3 and [Ngu07; Kös13]. To do so, we allocate an array of bodies (storing position and mass) in shared memory such that each thread can store a single body asynchronously. After loading a single bunch of all bodies into the group, we can continue processing by iterating over all bodies stored in shared memory. This process can be repeated until all bodies have been processed.

Note that this simplified algorithm does not take any edge cases into account where $((N-1) \bmod \text{group size}) \neq 0$. In such a case, this algorithm would perform out-of-bounds accesses that can be avoided by padding of the input buffer. There are also a variety of additional performance improvements that can be applied to this blocked algorithm. Common techniques include *loop unrolling* and more memory-access pattern optimizations by implementing *grid-stride* loops [KGK19a; KGK20b].

Algorithmic Considerations

Besides improving performance of the parallelized direct $O(n^2)$ algorithm, it is also important to consider conceptual algorithmic improvements. The presented improvements enable running the n-body simulation efficiently on reasonably sized data sets on a single GPU. If the data set size increases, the complexity of the algorithm reaches its limits. In such cases, reevaluating the actual algorithm being used and exploring alternative approaches is the only way to achieve improved performance.

Sticking with our sample, Barnes and Hut [BH86] introduced a hierarchical algorithm with complexity of $O(n \log n)$. There have been many parallelized versions of this algorithm, including GPU variants in the past. They considerably outperform the presented (even improved) version of the naive $O(n^2)$ algorithm on larger data sets. This is achieved by a hierarchical decomposition of the problem domain and propagating temporarily accumulated forces to regions in simulation space.

Introducing hierarchical space subdivision is a fundamental change to the initially discussed algorithm. Although the basic idea of computing forces between all bodies remains unchanged, this innovative algorithmic advantage allowed to overcome scalability and performance limitations. Our contributions throughout this thesis adhere to the same high-level principle: We realize performance improvements by introducing conceptual algorithmic advancements that go beyond merely leveraging implementation-specific features. These novel algorithms are designed with GPUs in mind, allowing them to utilize all the unique capabilities of modern GPU architectures.

Practical Considerations

For practical reasons, each property in $\mathbb{R}$ is usually represented by a `fp32` or `fp64` (a single/double precision floating point number with 32 or 64 bits) depending on the simulation requirements. However, floating-point operations cannot be considered to be commutative on actual hardware [NVI23a]. This can easily lead to visible artifacts when visualizing the underlying simulation data or can even make the whole simulation unstable [KK16]. Especially parallel implementations need specific care when precision is the primary concern. An alternative to floating-point numbers can be fixed point numbers based on integers to overcome the commutativity problems of their floating-point counterparts.

When performance is the primary concern of the simulation, accuracy is often less important. A simple and efficient way to improve performance is to choose large time-step sizes for Δt in favor of small ones. *Stability* is also very important as it expresses the desired high-level property of not causing invalid states that violate the fundamental constraints of the simulation [Mül+07; KK16]. In this scope, choosing Δt can easily become a challenging problem: An inappropriate value causes visual artifacts in the best case and makes the simulation unstable in the worst case. However, this varies depending on the algorithm and the implementation being used (see also Chapter 4).

3.3 Initial Work

One of our initially published projects was a combination of computer graphics, simulation, and HCI. It is outlined in the paper *"Gravity Games - A Framework for Interactive Space Physics on Media Facades"* [KSG15a], which describes a simulation framework for *m*edia facades. The underlying algorithm is based on an specifically optimized version of Algorithm 3 using several data structures based on the work by Barnes and Hut [BH86]. The basic idea behind this framework is to enable users to interactively explore and experiment with gravitational forces. These forces and their individual influence on each object are too complex to be predicted by humans. This becomes even more challenging in complex situations involving many objects that cannot be easily counted for observers. Therefore, the framework allows us to simulate thousands of objects in parallel using a GPU-accelerated simulation. It also features a specifically designed rendering pipeline based on on-the-fly generated geometry for displacement mapping and image-based alternatives like parallax mapping [POC05] (see Figure 3.4). It is especially optimized for large screen resolutions, multiple rendering devices, and different physical configurations of the displays/projectors being used.

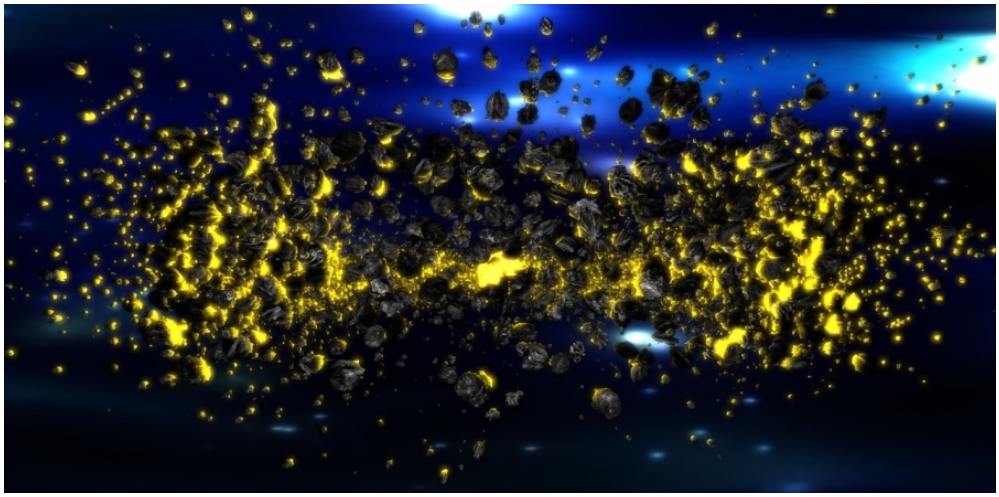

Figure 3.4: Sample rendering of a visualized asteroid belt that circulates around a light-emitting gravitational center [Kös13].

Figure 3.5: Media facade at the HBKSaar [KSG15a].

Our primary use case was the media facade at the *Academy for Fine Arts and Design* (*HBKSaar*) (see Figure 3.5): It consisted of five different back-projection units that were mounted behind the windows and projected onto a curtain. Due to the physical layout (around the corner of the building), it allowed observers to perceive a 3D like effect when viewing content from a specific point and angle. To benefit from this unique feature of the facade, the virtual camera location had to be adjusted accordingly during rendering (see Figure 3.6). In addition, the finally rendered images had to be distorted according to a displacement mask (also denoted projection mapping, Figure 3.7).

Figure 3.8 shows a photograph of our work deployed to the targeted media facade. Besides this facade, we have created several applications based on the same framework. In addition, we used this framework to realize a set of interactive media installations with different high-resolution projectors and tracking environments.

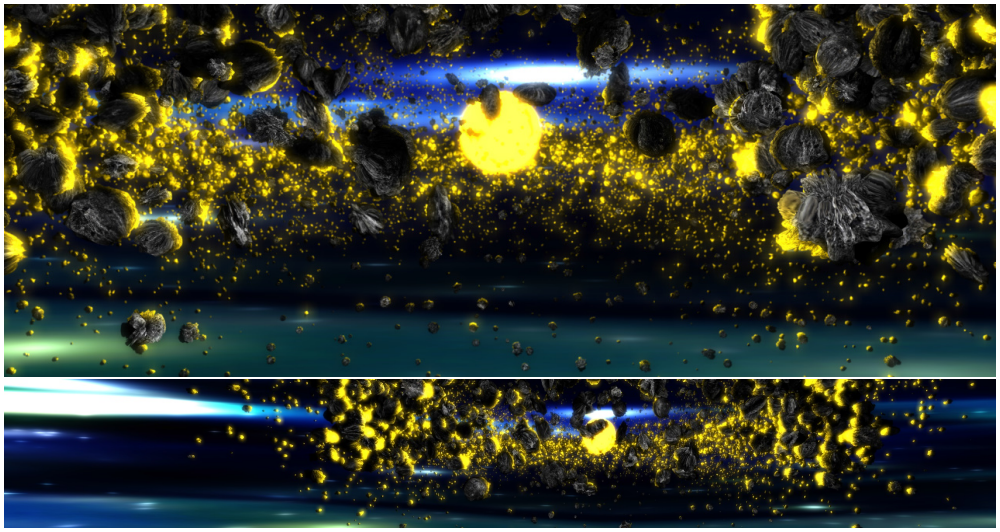

Figure 3.6: A sample rendering using a proper virtual camera setup to perceive a 3D like effect on the media facade [KSG15b]. Top: Main region of the simulation. Bottom: Whole rendering for the media facade projector setup [Kös13].

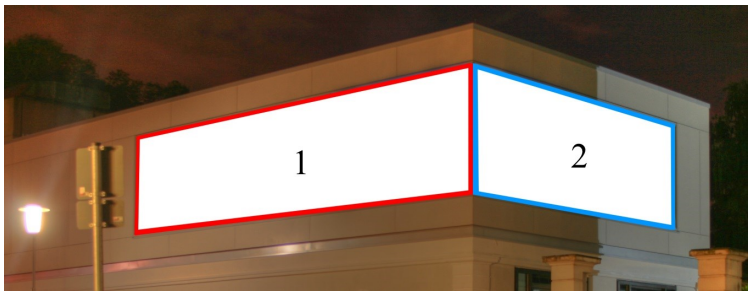

Figure 3.7: Sketch of the displacement mask used for projection mapping onto the facade [KSG15a; KSG15b].

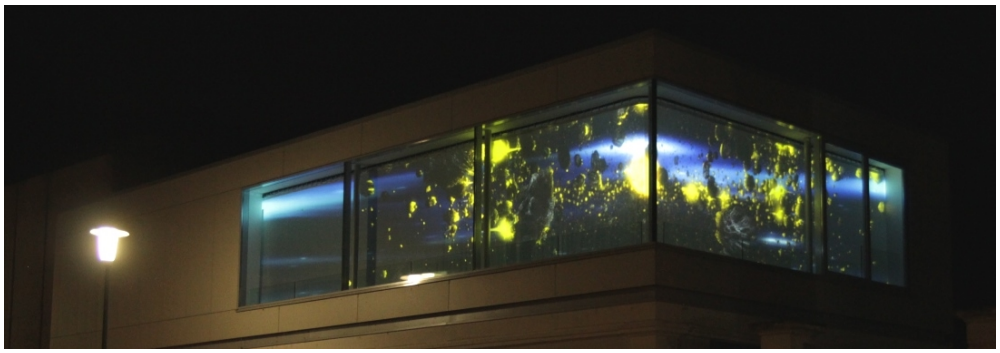

Figure 3.8: Picture of the deployed application [KSG15b].

A sample setup that was also published at an entertainment conference was the *Asterodrome* project. This project realized additional gravity simulations presented in as an interactive installation at the HBKSaar [Kös+15]. Figures 3.9 and 3.10 show exemplary renderings of the projector and the interaction-space concept. An image of the finally realized application can be seen in Figure 3.11.

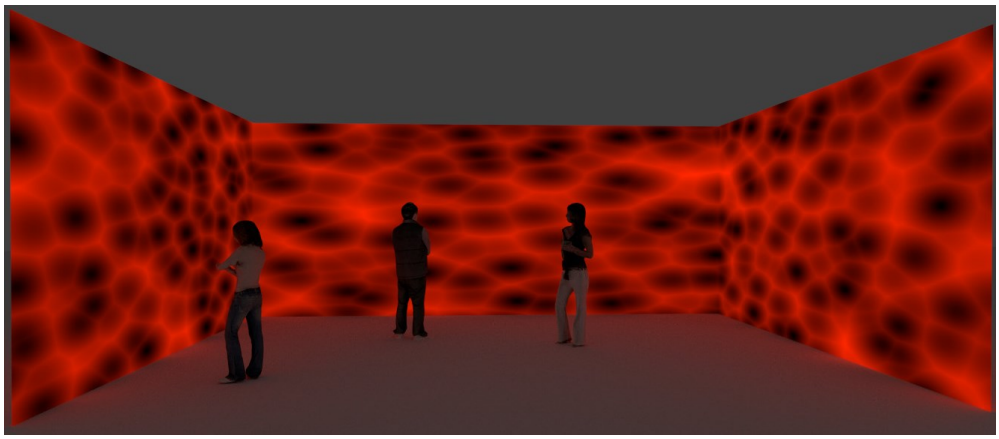

Figure 3.9: Conceptual sample rendering (1) of an interactive installation setup at the HBKSaar [Kös+15].

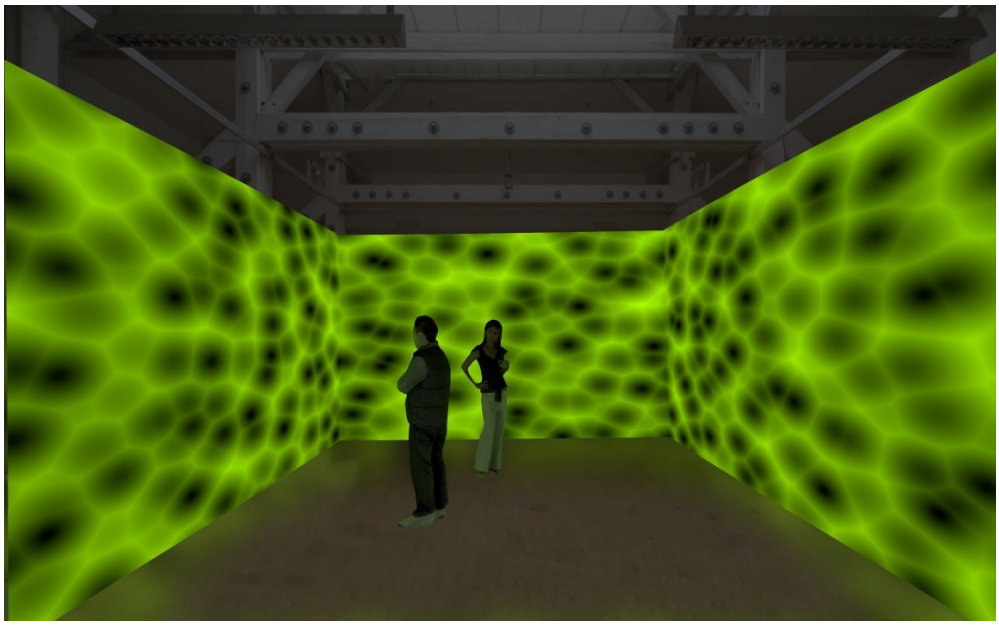

Figure 3.10: Conceptual sample rendering (2) of an interactive installation setup at the HBKSaar [Kös+15].

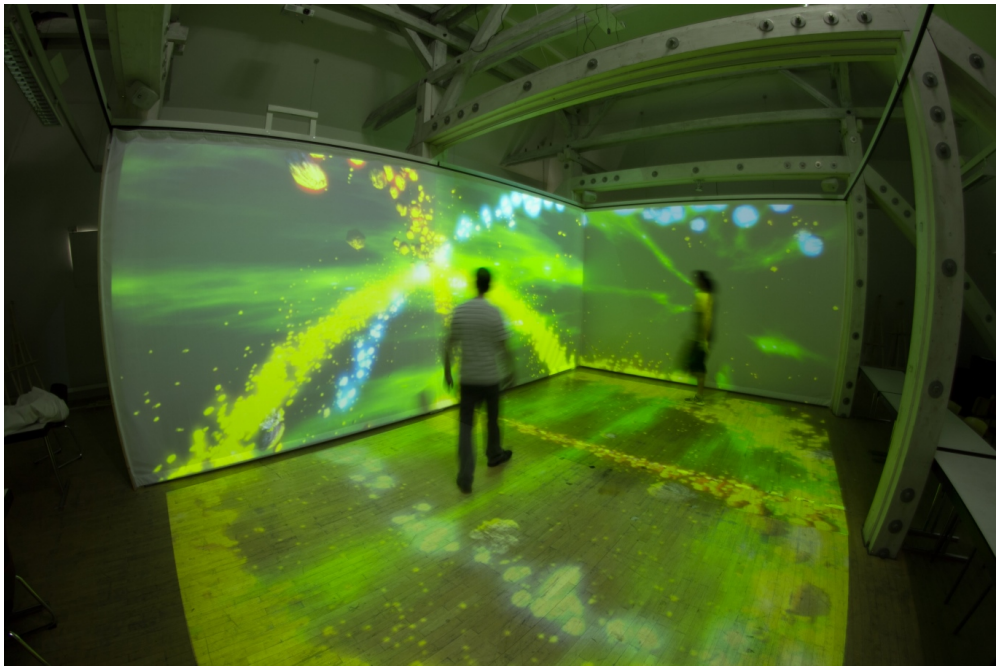

Figure 3.11: Picture of the *media theater* at the HBKSaar [Kös+15].

3.4 Contributions

After discussing related work in Chapter 4, Part I of this thesis makes the following major contributions:

- Chapter 5 presents two methods that can be used in scope of particle-based simulation processes. First, we introduce an adaptive fluid simulation model that can significantly improve runtime performance without introducing severe visual artifacts when rendering the simulated fluids. We then introduce a new method enabling users to interactively select potentially interesting subsets of large particle simulations/datasets in real time. Both approaches overcome known limitations with respect to scalability and runtime performance.

- Chapter 6 presents generic concepts and algorithms that are not limited to particle-based simulations. These key contributions can be applied to arbitrary GPU-based simulations to either improve runtime performance in general or to trade precision for performance. These methods have already been implemented and used excessively in production. See Part III for more information.

Pseudo-codes, illustrations, and sample code listings explain all methods in detail. All algorithms are designed to be implemented in any GPU-accelerated program using one of the programming languages available for this purpose.

3.5 Publications

The following list summarizes all contribution-relevant publications of this part of the thesis. Furthermore, contributions of the contributing author (*CA*) and all other authors (*CoA*) are explicitly listed. This helps to clearly separate own work that may be used in the dissertation from other contributions.

[KGK20a]	Marcel Köster et al. "High- Performance Simulations on GPUs Using Adaptive Time Steps." In: *20th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2020)*. Springer, 2020
CA	Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking
CoA	Feedback on the paper, paper refinement, implementation and benchmarking
[KGK20b]	Marcel Köster et al. "Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction." In: *International Journal of Parallel Programming* (2020)
CA	Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking
CoA	Feedback on the paper, paper refinement, implementation and benchmarking
[KGK19b]	Marcel Köster et al. *Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction*. 12th International Symposium on High-Level Parallel Programming and Applications (HLPP-2019). 2019
CA	Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking
CoA	Feedback on the paper, paper refinement, implementation and benchmarking

[KK18]	Marcel Köster and Antonio Krüger. "Screen Space Particle Selection." In: *Proceedings of the Conference on Computer Graphics & Visual Computing (CGCV-2018)*. The Eurographics Association, 2018
CA	Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking
CoA	Feedback on the paper, paper refinement

[KK16]	Marcel Köster and Antonio Krüger. "Adaptive Position-Based Fluids: Improving Performance of Fluid Simulations for Real-Time Applications." In: *International Journal of Computer Graphics & Animation* (2016)
CA	Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking
CoA	Feedback on the paper, paper refinement

CHAPTER 4

RELATED WORK

This chapter summarizes related work from the field of associated simulations, basic simulation methods from relevant domains, and selection algorithms. It is meant to provide a high-level overview of related approaches and algorithms that are most similar to our contributions, as well as their limitations. With this in mind, we focus only on the limitations that have been relaxed or even completely removed by our contributions in this work. Each section also includes paragraphs summarizing key aspects of related work, as well as the main differences to our approaches.

Please note that this list of publications cannot be exhaustive by definition, as we have made contributions to fields that touch on a wide range of domains.

4.1 Particle-Based Fluid Simulations

Section 3.1 introduced the basic concepts behind particle-based simulations on GPUs by using a gravity simulation as a running example. As two methods presented in this thesis rely on improving performance of fluid simulations in particular, this section introduces the basics and related work in this area. Fluid simulations often rely on the method of *smoothed-particle hydrodynamics* (*SPH*) to approximate smoothed quantities over a discrete set of points (particles) [GM77; Luc77; Mon92; NP94]. In this context a quantity A_i (a scalar property to be approximated) of the ith particle is defined using a smoothing kernel W (with finite support) around the current particle position via a convolution using W:

$$A_i = \sum_j \frac{m_j}{\rho_j} A_j W(||p_i - p_j||, h), \tag{4.1}$$

where A_j is the property value of the jth neighboring particle, m_j is the mass of the jth neighbor, and ρ_j is the local density in the current neighborhood

of the neighboring particle. Furthermore, p_i and p_j are the positions of the current (the ith) particle and p_j is the position of the jth neighbor. The most important property (besides the choice of W) is h, referred to as the *smoothing length* and/or the *smoothing radius* [Mon00; KK18], as it defines the length/radius to realize the convolution applied. Our method presented in Section 5.2 leverages SPH to compute local densities in real time allowing us to realize highly efficient particle selection processes.

Müller et al. [MCG03] realized an interactive fluid simulation by leveraging SPH. They used SPH to determine local fluid-density information (see also Section 5.2) by introducing custom weighting kernels W satisfying their needs. They built upon work by Desbrun and Gascuel [DG96] to compute densities using computed pressures. Having this concept at hand, they calculated correction (force-based acceleration) vectors that adjust particle positions to move them to their "intended" position. Similar to our gravity simulation example before, Müller et al. implemented their simulation pipeline so that the SPH-based quantities, force vectors, and position corrections are calculated in each simulation time step. Most importantly, this was in a real-time capable way. In [Mül+05], Müller et al. extended this work to support fluid-fluid interactions.

Unfortunately, their approach suffers from compressibility issues regarding the fluid(s) being simulated [BT07]. Compressibility refers to the fact particles are closer together than they should be in order to approximate an incompressible fluid. In other words, the simulated fluid volume may appear "squeezed" or smaller than it should be. This causes visual artifacts and considerable simulation deviations from a ground truth [Pre+03; BT07]. In order to tackle this problem, it is possible to use approaches like the one by Premoze et al. [Pre+03]. They introduced a method using an iteratively solved pressure Poisson equation formulation [Pre+03]. However, such methods were computationally expensive [BT07].

Becker and Teschner [BT07] introduced *weakly compressible SPH* (*WCSPH*) that allowed for small local density deviations in favor of computationally expensive solver iterations. The main downside of this approach were restricted simulation time-step sizes, which made more simulation steps necessary compared to other approaches [SP09; Ihm+14]. Solenthaler and Pajarola [SP09] came up with the concept of *predictive-corrective incompressible* (*PCISPH*). It relaxed previously constrained time-step sizes and achieved excellent results compared to *WCSPH* while improving performance significantly [SP09]. More work in this area was done by Ihmsen et al. [Ihm+14], who improved again on constrained time-step sizes and runtime performance using their novel approach called *IISPH*.

Meanwhile, Müller et al. [Mül+07] continued their work and built real-time capable physics simulations, not primarily intended to be used for the simulation of fluids but rigid and soft bodies. They introduced the concept of *position-based dynamics* (*PBD*). The idea behind this method is to simulate the physical mechanics of objects based on constraint functions, more precisely

nonlinear functions. Behind the scenes, these constraints are used to define an optimization problem that is iteratively solved via *constraint projection*. PBD leverages constraints represented by equations and in-equations of the form [Mül+07; Mül08]:

$$C_i(\mathbf{p} + \Delta\mathbf{p}) = 0 \text{ and } C_i(\mathbf{p} + \Delta\mathbf{p}) \leq 0, \tag{4.2}$$

where C_i represents the ith constraint. From a theoretical point of a view, each constraint enforces a certain property and is defined over a vector of all positions (particles) $\mathbf{p}$. The overall purpose is to find a position-correction vector $\Delta\mathbf{p}$, which can be added to the current positions $\mathbf{p}$ to meet the constraint. To solve for all position corrections $\Delta\mathbf{p}$, we use the following approximation [Mül+07; Mül08; KK16]:

$$C_i(\mathbf{p} + \Delta\mathbf{p}) \approx C_i(\mathbf{p}) + \nabla_{\mathbf{p}} C_i(\mathbf{p}) \cdot \Delta\mathbf{p}. \tag{4.3}$$

Restricting the solution to position corrections lying in the gradient direction ∇C_i and using the particle masses as weights, we get [MM13; KK16]:

$$\Delta\mathbf{p} = \lambda w_i \nabla_{\mathbf{p}} C_i(\mathbf{p}), \tag{4.4}$$

with a scaling factor λ and w_i being the inverse mass $\frac{1}{m_i}$ of ith particle in the simulation domain. With the help of this modeling approach, even sophisticated physical simulations can be realized in a mathematically appealing way while providing an excellent robustness and stability throughout the simulation [Mül+07; Mül08; MM13; Mac+14; KK16].

In order to model fluids to be used with position-based methods, Macklin and Müller [MM13] introduced the concept of *position-based fluids* (*PBF*) by leveraging previous work from Bodin et al. [BLS12]. They modeled a fluid density constraint of the form [MM13]

$$C_i(\mathbf{p}) = \frac{\rho_i}{\rho_0} - 1 = 0, \tag{4.5}$$

where ρ_0 is the fluid rest density to simulate and ρ_i the current density in the surrounding environment of the current particle. In order to realize this constraint, Macklin and Müller applied it to each particle in the simulation following again on previous work from Bodin et al. [BLS12]. This results in an evaluation of the equality constraint at the location of every particle in the simulation domain. In order to compute ρ_i, Macklin and Müller used the default SPH density estimator by Monaghan, described above [BLS12; MM13; KK16]. One year later, Macklin et al. [Mac+14] published a follow-up work that fully unified PBF and PBD and improved over previous works [Rob+08; Aki+12]. We improved on modeling of fluids with PBF by adding adaptive sampling in order to improve performance in Section 5.1.

Summary and Main Differences to Our Work Historically, research efforts have primarily focused on fluid simulations that rely on the SPH model. Various approaches were well researched to model fluids for specific application domains. The modern method PBF has emerged as an effective solution for creating realistic fluid simulations within the scope of unified particle-based physics. PBF offers a balance between robustness, performance, and applicability to various interactive applications. Our approach to improve performance in this general domain is by introducing an adaptation model to PBF. This innovative approach allows for improved efficiency without compromising the visual quality of the fluid simulation which was not possible before in PBF simulations.

4.2 Adaptive Particle and Fluid Simulations

In order to improve performance of methods discussed in Section 4.1, *situation-aware* or *adaptive* methods have been very popular [KK18]. The basic idea is to automatically adjust the accuracy required to "successfully" run a particle/fluid simulation to specific criteria in order to reduce computational effort and thus performance. Successful here refers to properties such as (numerical) stability, visual appeal, and runtime. This section covers general adaptation approaches dedicated to the particle and fluid simulation domain.

An idea by Adams et al. [Ada+07] was to use different particle sizes in the simulation domain. They built their adaptive simulation model upon this concept by improving the simulation precision based on whether a visually important region was affected or not. Visual significance was determined using distances to rendered geometry in the simulation domain. Their approach resulted in a decrease in the number of particles required, as coarse-grained approximations were performed with larger particles, ideally replacing the motion of multiple smaller particles. Here, they used domain-dependent criteria on when to split larger particles into smaller ones and when to combine them. Reasoning about particles with different sizes in the same simulation requires a considerably more sophisticated solver method. Moreover, it would cause a fundamental change to commonly used simulation engines [Mac+14; KK18]. Consequently, we did not follow this research direction.

Based on the distance of particles to the surface of the fluid to be simulated, Hong et al. [HHK08] also used heterogenous particle configurations. In addition, they used multiple layers (four in their paper) with specific splitting and merging rules for particles. In their case, the actual grid was added to the grid-free particle simulation, making this contribution a hybrid approach. In a follow-up paper, Hong et al. [HHK09] improved on their method via reducing memory consumption of their layer idea by introducing adaptive grids to improve scalability. Yet, the overall adaptation methodology stayed the same. Zhang et al. [ZSP08] used a related method ultimately leading to domain-

specific split and merge operations based on certain conditions to improve performance. They also contributed a GPU-friendly realization for their method. As discussed in the previous paragraph, splitting and merging makes simulation logic more complex. Even worse, some methods of this kind require additional data structures causing scalability issues during parallelization on GPUs.

Conceptually different is the approach by Solenthaler and Gross [SG11]. The authors made the need for split-and-merge operations obsolete by maintaining two versions of the "same" simulation with different scales. Same means here that the model with the coarser-grained particles is conceptually the baseline model. While running the simulation, certain regions are selected which will be linked to regions of the more fine-grained simulation. As mentioned above, this eliminates the need for splitting and merging and simplifies the particle size considerations: Both versions of the simulation consist only of particles of the same size. An extension of this method improving on the number of detail levels was published by Horvath and Solenthaler [HS13]. In all of these methods, the simulation detail was chosen according to properties of the camera. Although these ideas are very appealing at first sight, they also require an adaptation of the underlying simulation basics. This would again cause severe changes to existing simulation engines limiting the applicability of our methods [Mac+14; KK18].

Goswami and Pajarola [GP11] published a method based on the idea to distinguish between inactive and active particles. Positions of particles in both categories were adjusted based on different criteria. The authors made the decision whether a particle is considered active or not based on the speed of a particle and whether it was close to the volume boundaries (e.g., the surface). They further improved on their method by assigning particles in certain regions while improving WCSPH [GB14]. Different regions were updated with different frequencies using adjusted time-step sizes assigned to particles in distinct regions. Related to that Ihmsen et al. [Ihm+10] extended PCISPH by adding adaptive time-step sizes. Our method presented in Section 5.1 is built on a similar idea to treat particles differently based on their importance. However, we do not statically differentiate between active and inactive ones in the scope of a whole simulation step and we do not adapt the time-step sizes. This is because adjusting time-step sizes requires an complete understanding of the whole simulation domain which makes interoperability with other solvers considerably harder. Instead, we perform fine-grained adjustments to the simulation quality per particle to maintain stability and preserve compatibility with our approach we approved upon.

Summary and Main Differences to Our Work Most publications focused on adaptive sampling to improve performance, by either using different particle sizes within the simulation domain, or choosing time step sizes adaptively. Choosing heterogeneous particle sizes to change the actual simulation

resolution requires adapting the underlying solver equations according to specific domain requirements. Applying this methodology to PBF would also require reconsidering all PBF mechanics. Since the idea is to realize an adaptation extension that integrates with existing unified PBD solvers, this would ultimately not be directly possible. It would require changing the representation of PBF particles and introducing a sophisticated coupling mechanisms between our adaptive PBF world and the remaining parts of the surrounding PBD-based solver.

Instead, we followed the general concept of choosing time-step sizes adaptively for PBF using visually guided heuristics presented in Section 5.1.2. Using such heuristics is based on prior work which described adaptation models using virtual camera properties and physical objects in the simulation domain. In contrast to these works, we introduced a new method to realize adaptive time steps for PBF simulations.

4.3 Improving Utilization of Generic Parallel Simulations

Improving performance of parallel simulations built implicitly or explicitly on the concept of logical building blocks (like *rules*, see Section 6.2) is widespread and well researched over the years. Of particular interest to our research are publications in the area of parallel computing that exploit certain hardware properties that apply to generic parallel simulations. Many algorithms discussed in this section rely on the high-level idea of *compaction*. Compaction here always refers to filling "gaps" in a sequence of elements, which may be a data stream (usually referred to as *stream compaction* [BOA09]) or a computation/instruction stream (referred to as *thread compaction*, see Section 6.2.1 and Figure 6.6 for a detailed explanation of thread compaction). Billeter et al. [BOA09] published a hardware-aware method based on prefix sums to realize thread compaction on NVIDIA GPUs. In short, the idea of thread compaction is to move threads (including their register data) to leverage unutilized processing units/circuits on the chip. This particularly includes arithmetic/logic units.

A good example for that is the work by Fung and Aamodt [FA11] who made use of thread compaction to overcome performance degradation in the presence of divergent control flow. Their contribution was an algorithm operating on the hardware level making software overhead unnecessary. Unfortunately, this algorithm cannot be easily transferred completely to arbitrary hardware, leaving room for software optimizations. In addition, domain-specific optimization in software can always be added to improve performance using domain knowledge, as done in our case.

Similarly, Rhu and Erez [RE13] maximized GPGPU utilization using CUDA for NVIDIA GPUs by analyzing control-flow graphs. They used permutations

to reorder threads in software and discussed certain permutation strategies. Although their method is more generic than our method presented in Section 6.2, we can take advantage of our domain knowledge to improve performance in our use cases (e.g., multiple optimizer states). Conceptually, their method is most similar to our approach if we focus on the compaction contributions. Unlike them, we also contribute a special memory layout that fits seamlessly into our compaction approach.

Most similar to our approach are also well-known methods from the fields of computer graphics and visualization. Hoberock et al. [Hob+09] used thread compaction to reduce divergence in shaders during rendering in the context of deferred shading. They relied on prefix-sum computations in global memory to rearrange the require data before rendering. Highly similar to that is the contribution by Hughes et al. [Hug+13] to improve performance in the data visualization domain. However, they did not perform the compaction step completely in global memory as they used group-wide compaction and synchronized data accesses across all groups using atomics. Wald [Wal11] also performed compaction in group-wide accessible shared memory to achieve considerable performance improvements in the path-tracing domain. Our approach also relies on thread compaction in shared memory. All threads in a group collectively contribute to the compaction step before beginning the actual work. In contrast to these approaches, we target the domain of simulating multiple world states in parallel that also require specific memory layouts to benefit from such optimizations.

Summary and Main Differences to Our Work The history of research on thread compaction to improve utilization of SIMD and SIMT hardware is extensive. Various generic approaches have been proposed on both the hardware and software levels, focusing on addressing the problem from general perspective.

In the context of massively parallel simulations, our primary concern is to also parallelize across multiple states. This can be viewed as running multiple simultaneous simulations that should fully exploit parallel computation capabilities available on GPUs. By considering this specific target domain, we have developed an approach based on thread compaction for enhanced performance in our particular context. Additionally, we have made a significant contribution by proposing an optimal memory layout that further enhances the overall performance within our domain setting.

4.4 Adaptive Time Stepping for Generic Simulations

The general concept of adaptive time stepping itself has also been very well researched for years in a huge variety of different domains [Kay+10; Car+10; GH13; PB15] (see also Section 4.2). Prominent examples are from the domain of solving partial differential equations [GH13; KGK20a] or particle simulations [HS13; KK18]. Many of them also make use of interpolation functions to approximate (and predict) intermediate values to realize adaptive time stepping [Ihm+10]. All presented papers so far made use of domain-specific knowledge to overcome time-step (size) restrictions by adding specially designed data structures or adjusting the underlying simulation mechanics (e.g., the work by Adams et al. [Ada+07] or by Ihmsen et al. [Ihm+10]). While also being from the area of fluid-based simulations, a recent work by Mayr et al. focused on more iterative calculations of time-step sizes [MWG18]. This work is also worth mentioning because of their proposed generic way to determine time-step sizes based on error calculations. The time-step sizes are calculated in a first step and applied to the simulation in a second step.

Our contributions in this area focus on improving performance of generic parallel simulations in discrete object-oriented simulation domains that exist in the scope of optimization solvers (see Section 6.3). In this scope, our basic concept is built on related work to use interpolation functions in order to approximate intermediate values. The goal is to relax time-step restrictions of arbitrary simulations following certain design principles. Comparing our approach to the work by Mayr et al. reveals that we also estimate time-step sizes for each of our simulation components beforehand. Afterwards, we try to find a compatible step size that will work for each component in every time step. However, we are not focused on fluid simulations and focus on optimization states instead which causes domain-specific considerations from their work to be non-applicable to our problem space.

Most related to our approach is the method by Garcia et al. [Gar+11]. They introduced GPU accelerated and CPU managed parts of their application, while the parts running on the GPU implement the actual simulation logic. In each simulation step, they performed a time-step estimation phase on the GPU in parallel first for each of their GPU-accelerated simulation modules and transferred this value back to the CPU space. Once they had this value at hand, they could adjust the global time-step size for each module accordingly for the actual simulation step afterwards. Our method is very similar to the one described here in terms of using an initial time-step estimation phase across all modules involved. Moreover, we also rely on synchronization of our determined time-step sizes on the CPU side to share and propagate time-step information. In contrast to this approach, our method supports dynamic interpolation between intermediate results to relax time-step restrictions. Our design conceptually improves over related work significantly.

Summary and Main Differences to Our Work Adaptive sampling and time-stepping techniques have been widely studied while focusing on partial differential equations and specific application areas (e.g., fluid simulations). In contrast, our research targets adaptive time stepping for (agent-like) object-oriented simulations meant to be used in the scope of optimization processes. Related work has addressed time-step-size restrictions by interpolating intermediate values using problem-dependent interpolation functions for particular use cases. Moreover, several papers introduced GPU-based adaptive time stepping for arbitrary domains, without paying attention to interpolation capabilities to relax time-step sizes. Our method in this scope combines these worlds while combining relaxation of time-step restrictions and GPU-driven acceleration.

4.5 Particle-Based Selection

In the fields of visualization and HCI, prominent tasks are object selections. This includes selecting multiple objects at the same time or focusing on a single one [DFK12]. Moreover, much research has been done in the field of object selection in 3D environments [Min95]. In these settings, common methods are based on ray casting and/or the use of projected image planes to select objects in a rendered scene [Min95; Pie+97]. This had been extended to virtual reality environments [Val+10], as well as touch and pointing based selection methods using stereoscopic displays [DFK12; Dai+14]. However, all of these methods focus on the selection on a few objects (often a single one, for example) while enabling that in either 2D and/or 3D domains.

Since our method targets the domain scientific visualization, we are interested in analyzing and handling datasets consisting of several hundred thousand particles (or data points, see Section 5.2). In contrast to previously mentioned methods, this domain requires selection methods operating on 3D volumes. Different technologies had been published in this area [KK18]. A prominent method in this area is the one by Steed and Parker [SP04] who used cone tracing to realize the actual selection process. In their work, they selected either all objects intersected by the cone or the object closest to the cone center. Considerable improvements were made over the years by introducing *structure-aware selection methods* [WVH11; Heg+12]. Such approaches rely on underlying analyses to reveal more information about the dataset being used. Subsequently, this data is used for the actual selection inference, which determines which part of the dataset should be selected.

Very sophisticated and precise methods (which are also very similar to our approach) are those of Yu et al. [Yu+12; Yu+16]. The first methods presented by the authors were *CloudLasso* and *TeddySelection* [Yu+12]. Both selection algorithms are based on a selection lasso drawn in 2D by a user. We focus on *CloudLasso* because of its relation to our method and it was determined to

have superior selection quality and runtime performance on their evaluation scenarios. It relies on a density estimation method to determine local particle densities (similar to SPH-based methods, introduced in Section 4.1).

After having calculated density information, the selection process is realized using a 3D grid resulting in a 3D selection mesh. During this process, they used the *Marching Cubes* [LC87] algorithm to convert their intermediate grid data structure into meshes (see also Section 5.2). In sum, this concept allowed them to map a 2D input lasso to a 3D selection volume using their algorithms.

The authors further improved their selection algorithms and invented the family of *CAST* algorithms [Yu+16], where *CAST* refers to *context-aware selection techniques* [Yu+16]. In analogy to their *CloudLasso* method, all CAST algorithms were built on top of a dataset analysis phase for each selection step. All methods used the same very expensive density-estimation method and are built around the high-level notion of *clusters* in the input particle datasets.

As for the clusters, each of them represents a 3D region in space comprising particles of a certain density. The idea was that a cluster of particles can be easily visually identified and separated from other parts of the dataset (by humans), and thus selected by users [Yu+16]. Based on their previous experience, they focused on different use cases, which they covered with specially adapted methods. They introduced the following algorithms for their application scenarios [Yu+16]:

- *SpaceCast* selects (sub)clusters considering the shape of the lasso,
- *TraceCast* selects only whole clusters, but does not require very precise lassos, as it operates at a very coarse-grained level,
- *PointCast* selects whole clusters using a single mouse click/touch interaction and a selection heuristic.

Unlike *PointCast*, our approach is inherently lasso-based, which means that we require the selection lasso shape to infer the intended choice by the user (Section 5.2). Moreover, we do not reason about clusters at all in the dataset as we are able to map the selection to (sub)space regions of the dataset without intermediate data structures.

An essential part of their algorithm was again based on a custom particle-density estimator. They relied on a modified version of the *Breiman density estimator* approach [B J+11] combined with specifically chosen kernels [Yu+16]. Their choice was also motivated by related work by B. J. Ferdosi et al. [B J+11] who compared different density estimators for astronomical datasets. The method is based on a uniform 3D grid containing smoothed density information and requiring $O(m^3)$ memory, where m represents the number of grid cells of each dimension. Actual density information per particle is then determined using linear interpolation between neighboring grid cells. Due to their choice of the density estimator and use of the Marching Cubes algorithm,

scalability was a limiting factor in terms of memory consumption and runtime performance.

Similar to the algorithms of the CAST family (and to previous work by Yu et al. in general), our idea is based on the same high-level concept. We also start with a 2D selection mask that is drawn by a user and map this information to the "intended" selection (see Section 5.2 and [KK18]). The overall aim of our method is to considerably improve runtime performance and reduce memory consumed. By overcoming these limitations, it is also possible to use our methods in rapidly changing dataset environments in which data may even come from a real-time simulation running in the background rather than a statically loaded dataset.

Particularly related to their methods is our use of a density-estimation based selection process. We use local density information to realize selection-lasso mapping to particles. As mentioned above, their choice of the density estimator was already motivated by related work. In order to significantly improve performance of this expensive step, we propose a density-estimation algorithm directly based on SPH, which was developed for astrophysical problems [Luc77; Mon92]. Moreover, SPH methods have been successfully applied to particle-based real-time applications (like the ones by Macklin and Müller [MM13] and Köster and Krüger [KK16]). The limiting factor in terms of performance in this case is the iteration over neighboring particles. However, this allows us to avoid expensive 3D grids at all using recent work by Groß et al. [GKK19; GKK20].

Summary and Main Differences to Our Work Besides well known publications in the HCI space around object and multi-object selection using traditional input devices (e.g., keyboards and mice), we focus on data sets with millions of points to be selected. In this context, state-of-the-art selection methods are the algorithms from the CAST family. The work on point (or particle-based) data sets using specifically designed deep volume analyses (complex analyses operating on the data set directly) to map user input to intended selections. Unfortunately, these methods suffer from runtime and memory-consumption issues, which limit their scalability when dealing with larger data sets. Our approach overcomes all these limitations by enabling real-time feedback and excellent scalability.

CHAPTER 5

IMPROVING PERFORMANCE OF PARTICLE-BASED SIMULATION AND SELECTION PROCESSES

This chapter presents contributions to improve performance of particle-based simulation and selection processes. The first section introduces general simulation basics using position-based and SPH-based simulations. Section 5.1 presents an extension to position-based fluid simulations that improves performance without causing significant deviations in terms of the quality of the rendered simulation. Section 5.2 shows a method built on SPH-based kernels that allows users to select particle subsets in the scope of large scale datasets in real time. As a first motivation, Figure 5.1 shows a rendered particle-based gravity simulation using the tool from Section 5.2.

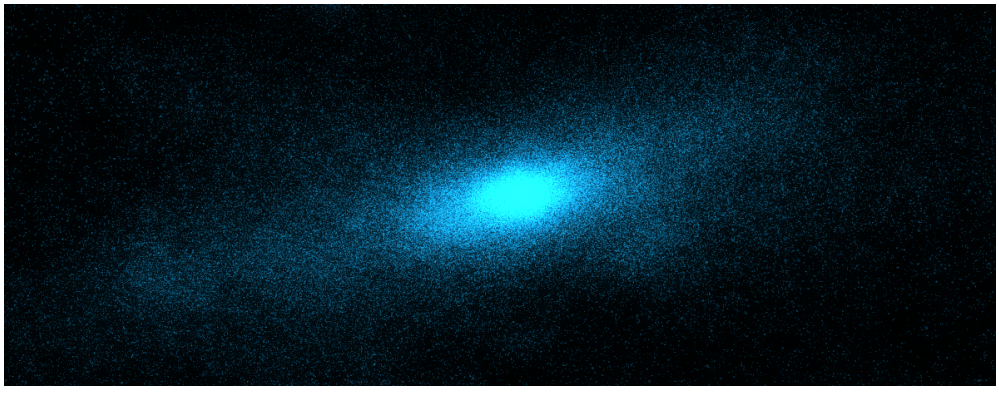

Figure 5.1: Demo gravity simulation rendered with our selection tool from Section 5.2.

5.1 Iteration-Adaptive Position-Based Fluids

PBF is one of the most modern methods, which even fits seamlessly into the context of globally unified particle physics [Mac+14]. The approach of unified physics enables developers to rely on a generic, multi-purpose, and reusable physics-simulation pipeline that can be used in state-of-the-art computer games, for instance. Games in particular (especially open-world games that use large simulation domains) make PBF integration challenging due to the computational resources required. The original PBD approach used a Gauss-Seidel solver to solve the different constraint functions. Gauss-Seidel iterations imply a sequential execution in which the position-correction updates are computed and applied. This way, position updates are directly visible to all other updates and are affected by additional constraints. The sequential execution order can be circumvented by using Jacobi-style iterations for solving the presented density constraint [MM13]. This allows parallel computation and application of position corrections. Since the corrections in each iteration are propagated only to the immediate neighbors, the convergence rate decreases as the number of particles increases: A larger number of iterations causes the density constraint to be approximated more precisely, which in turn reflects the degree of the approximated incompressibility.

Therefore, one of the major bottlenecks in the scope of PBD-based simulations are the solver iterations I: The more iterations we need to perform, the longer a full simulation step will take. The same holds true for PBF simulations that share the same solver concept based on constraints. As outlined in Chapter 4, there have been approaches trying to experiment with varying particle sizes and with the underlying simulation algorithms. Unlike all these prior approaches, we present our method of *Simulation Adaptive Position-Based Fluids*, or short *APBF*. Our approach does not weaken fundamental simulation properties which makes it a perfect choice for interactive applications that rely on visual quality instead of actual simulation precision. Furthermore, it can be seamlessly integrated into any PBD based simulation system.

The main idea of APBF is to adjust the solver iterations based on environmental conditions and apply constraint effects adaptively [KK16]. This is possible in general without causing severe simulation issues due to the robustness and stability properties of the underlying PBD method. Our approach in this scope is to solve the density constraint in such a way that we are still able to achieve high-quality visual results while improving the performance significantly. Figure 5.2 shows a visual comparison of our approach to the PBF method using a water-like (upper row) and an oil-like (lower row) fluid. In the presented use cases, APBF performs up 77% faster compared to PBF.

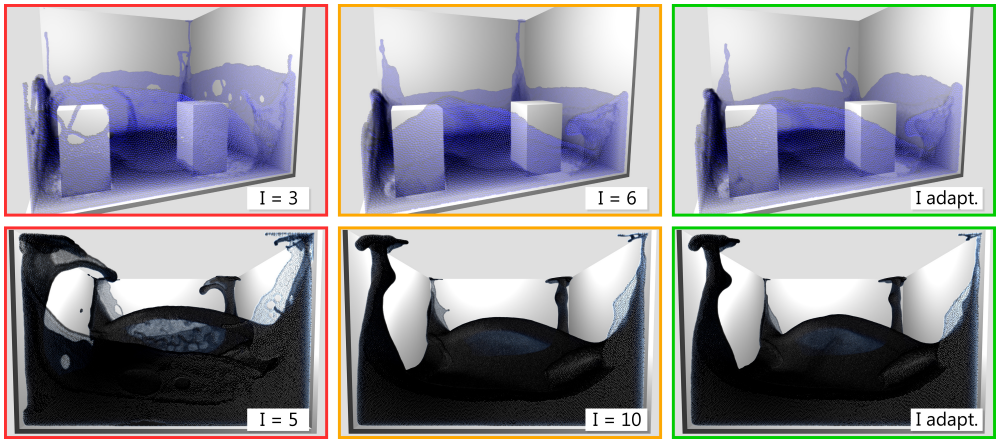

Figure 5.2: A visual comparison of a rendered water-like and an oil-like fluid simulated with PBF (red and yellow) with a fixed number of solver iterations and APBF (green) that uses an adaptive number of solver simulations ($I \in \{3, \ldots, 6\}$ top and $I \in \{5, \ldots, 10\}$ bottom, extended version of the teaser image published [KK16]).

Since the number of solver iterations directly influences the approximated incompressibility, we need a fine-grained approach to adjust the number of iterations on a per-particle basis (see Chapter 4). To do so, we use specially designed *classification* (CL) information that is computed per particle. The CL information allows us to differentiate between more important (interesting in terms of the visual quality of the simulation) and less important particles. Moreover, we can directly map this information to the number of iterations each particle will be affected in. The actual CL of a particle can be formally defined by [KK16]

$$cl(p_i) : \mathbb{N} \to \{1, \ldots I_l\} \subset \mathbb{N}, \tag{5.1}$$

where p_i refers to the ith particle and I_l to the maximum CL. The purpose of this definition is to ensure that all particles with a given classification are processed at the same time during a solver iteration. In addition, all particles with a CL that is higher than the current CL will also be considered. This is required to adjust all particle positions in the current iteration and in further iterations properly. Otherwise, particles in future iterations would not be adjusted in previous iterations, and thus, not receive any position corrections. This would lead to incorrect position approximations and ultimately to instability of the simulation.

In order to reason about considered and not-considered particles, we introduce the notion of *active* particles. A particle i is called active in the scope of iteration $l \in \mathbb{N}$, if its position has to be adjusted according to its CL (see Equation (5.1), [KK16]):

$$active(p_i, l) := cl(p_i) \geq l. \tag{5.2}$$

Note that according to these definitions, all particles in the first iteration $l = 1$ are always considered active. The set of all active particles in a certain iteration l can be defined using our notion of an *active* particle as [KK16]

$$P_l := \{\ p_i \mid active(p_i, l)\ \}, \tag{5.3}$$

where the particles that have already exceeded their number of iterations (in which they were moved) are then given by [KK16]

$$\hat{P}_l := \begin{cases} \emptyset & l \leq 1, \\ P_1 \setminus P_l & else. \end{cases} \tag{5.4}$$

Figure 5.3 shows a conceptual visualization of three different CL-dependent iterations to highlight which particles are affected. For simplicity, in this example we consider particles with a larger distance to virtual camera at the top right as less important. Particles with the largest distance will receive a CL of 1, whereas particles considered to be closest to the camera will receive a CL of 3. In the first iteration, all particles (CL ≥ 1) will be considered to be active and moved accordingly. Afterwards, particles with the lowest CL will not be considered any more since they have already reached their target position and remain frozen for the rest of the whole simulation step. Consequently, the second iteration will only move particles with a CL of 2 or higher, which only affects all top-most/right-most particles. In the last iteration, only particles with the highest CL of 3 will be moved; all other particles are already considered to be placed properly.

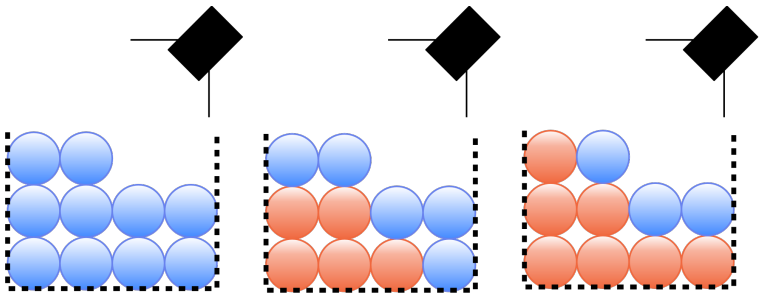

Figure 5.3: Conceptual rendering of three CL-dependent solver iterations [KK16]. First iteration with CL = 1 (left) to third iteration with CL = 3 (right). Particles in red are considered frozen and will not be moved in following iterations.

5.1.1 Density and CL Adjustments

As in PBF, density fluctuations can occur in APBF due to an inadequately approximated solution of the density constraint and hence the incompressibility property we wish to preserve. The previous sample already gave the intuition behind the underlying principle that particles with a higher CL can compensate density variations that may be caused by particles with a lower CL. In the worst case, our approximation is only as good as the minimum CL based on our per-particle information. At best, this is the number of iterations that would have given us the best results regarding visual quality and/or density. Figure 5.4 demonstrates how particles with a high CL can compensate coarse grained density approximations in their neighborhood during the iterative solving process. As outlined above, particles that have already reached their final position (based on their CL information) will not be moved anymore. However, they will still be considered to compute position adjustments for the ones that remain active: The active particles can still "respond" to all positions of their neighboring particles and adjust their positions accordingly.

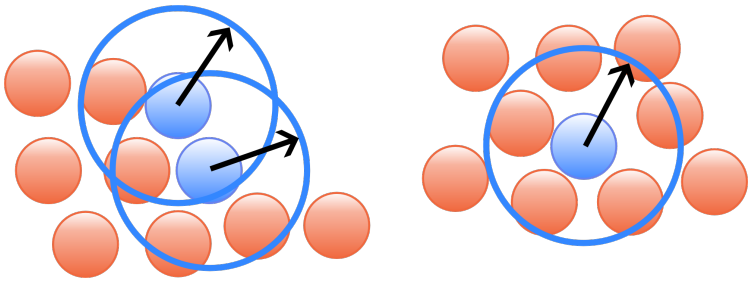

Figure 5.4: Different smoothing radii h around active particles (blue) that can compensate coarse-grained density approximations caused by particles with a lower CL (red) in their neighborhood [KK16].

Although we do not use the same CL for all particles, we are still able to achieve a consistent average density across the whole simulation domain using this approach (see Section 5.1.4). In order to ensure consistent densities in all cases, the classification assignment and the mapping to actual solver iterations have to satisfy certain requirements. First, each scenario/application domain requires a certain minimum number of iterations of the solver to achieve a reasonable overall fit to the density constraint. Note that this requirement also applies to PBF, in order to guarantee stability of the underlying simulation (e.g., a single iteration can lead to instability depending on the scenario). Second, the transition between different CLs has to be as smoothly as possible. Transition in this scope means that we need to smoothly distribute the CL assignments to avoid regions in which particles from "very different" CLs can interact with each other within the smoothing radius h. To be precise, the difference between CLs in the immediate neighborhood around a certain particle

should be ≤ 1 and is formally given by

$$\forall p_i \in \text{Sim } \forall p_j \in N(p_i, h) : |cl(p_i) - cl(p_j)| \leq 1, \tag{5.5}$$

where h is the smoothing radius, p_i the ith particle and $N(i, h)$ the neighborhood function that gets all neighboring particles within the radius h in the surrounding of p_i. This is necessary not only to maintain an average density, but also to minimize visible artifacts of the simulation. Consider Figure 5.5, which depicts a scenario in which the CL differences in the immediate neighborhood are significantly greater than 1. In this sample, particles with a high CL cannot compensate fluctuations caused by already fixed positions from particles with a low CL anymore without causing significant deviations to a ground-truth PBF simulation.

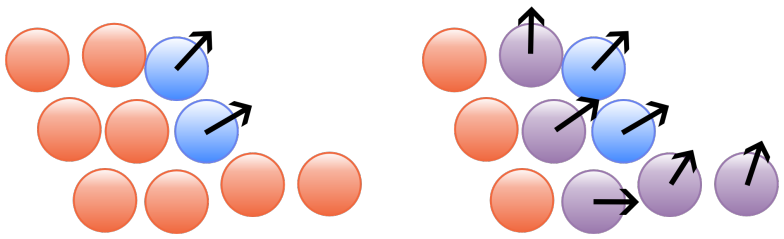

Figure 5.5: Example with big CL differences greater than 1 (left). The non-smooth CL distribution left suffers from much more compression artifacts compared to a smooth CL distribution on the right, which allows to compensate these artifacts in future iterations.

Due to the nature of the simulation, particles can be accelerated very strongly and thus reach high velocities. From this follows that it is possible to violate this constraint from time to time, since particles with a low CL can quickly move to regions with a high CL and vice versa, which relaxes Equation (5.5) to

$$\forall p_i \in \text{Sim } \forall p_j \in N(p_i, h) : |cl(p_i) - cl(p_j)| \leq 1 + \epsilon, \tag{5.6}$$

where ϵ is a configuration and domain-dependent value. The overall task is to minimize ϵ with respect to all particles. However, it is important to note that this inequality represents only a single snapshot. The actual task is to minimize ϵ across *all time steps* of the simulation:

$$\forall t \in [1, \ldots, T] \; \forall p_i \in \text{Sim } \forall p_j \in N(p_i, h) : |cl(p_i) - cl(p_j)| \leq 1 + \epsilon_t. \tag{5.7}$$

Since ϵ_t is time and simulation-state dependent, it is not feasible to pre-compute an optimal CL assignment strategy before the start of the simulation. Instead, we have to dynamically adjust the CL information to minimize each ϵ_t separately on-the-fly. In order to achieve this, we need to recompute all particle CLs dynamically in every time step of the simulation using heuristics to minimize all ϵ_t in Equation (5.7).

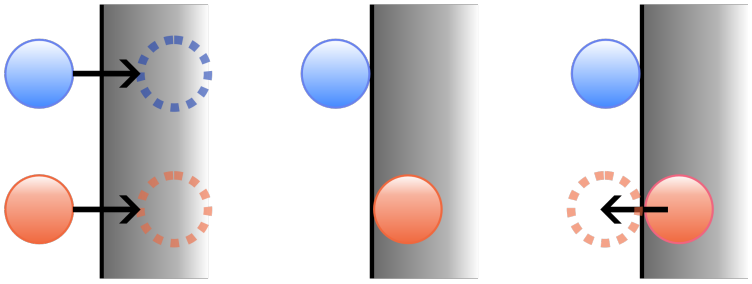

Figure 5.6: Two particles (a high-CL one in blue and a low-CL one in red, left) [KK16]. The velocity vectors (black arrows) of both particles point into the direction of the obstacles (gray). Due to different CLs, the coarse-grained approximation of the red particle might cause the particle to be moved very closely to the obstacle without adjusting its velocity vector properly. This in turn can cause the red one to be moved into the obstacle at the beginning of the next simulation step during the position-prediction step (center). Afterwards, the corrected position within the upcoming simulation step will adjust its position and the velocity vector (right), which leads to visible artifacts like jittering.

Macklin et al. outlined that they use the concept of pre-stabilization in their work to compensate coarsely approximated constraint solutions [Mac+14]. Figure 5.6 visualizes this concept in a PBD simulation in the context of particles being moved into obstacles. The basic idea of pre-stabilization is to solve specific constraints (mostly collision constraints) in a small pre-solver loop to move particles to positions, in which they will not seriously violate constraint-implied invariants. In addition, they will be moved to these newly adjusted positions without affecting their velocity. Otherwise, this could easily lead to particles getting a very high velocity, which in turn might cause other constraints to be violated. This can ultimately cause instability and visible artifacts. APBF also leverages this approach to circumvent such issues. Moreover, it is particularly important in our method to avoid clashes between particles of different CLs caused by particles with an extremely high velocity vector.

5.1.2 Adaptation Models

To compute CL information per particle, we introduce two different on-the-fly adaptation models: *distance to Camera* (DTC) and *distance to visible Surface* (DTVS). DTC computes the distance of every particle to the current camera position and gives a higher CL to particles closer to the camera (see Figure 5.7). DTVS pays more attention to the simulated fluid volume by computing the distance of each particle to the particle closest to the camera (the visible part of the volume, see Figure 5.8). As shown in both figures, we use linear interpolation to ensure a smooth CL distribution over all particles.

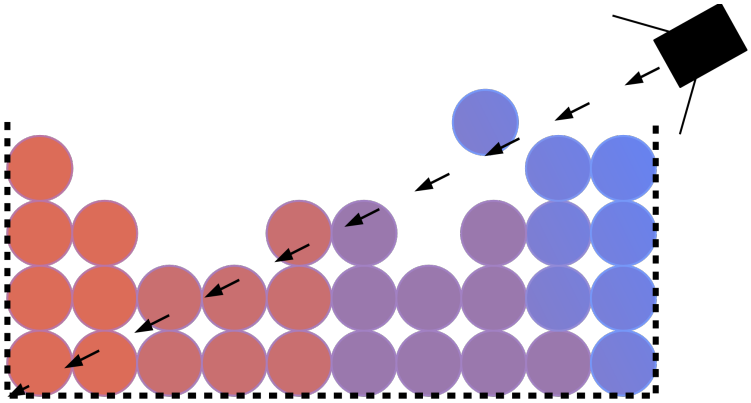

Figure 5.7: Conceptual visualization of the DTC adaptation model based on a virtual camera at the top right [KK16].

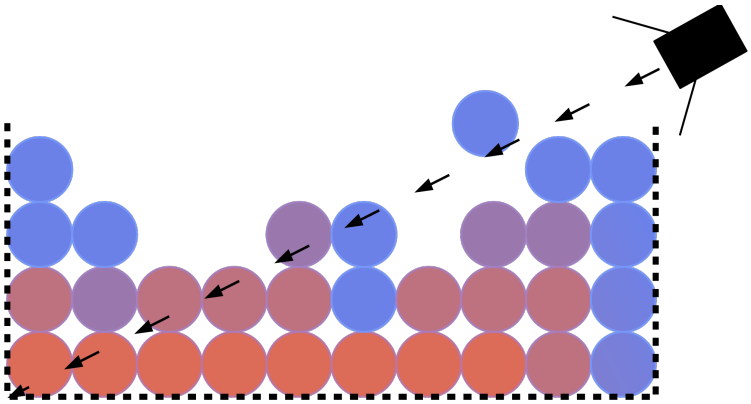

Figure 5.8: Conceptual visualization of the DTVS adaptation model based on a virtual camera at the top right [KK16].

Figure 5.9 shows the differences between both approaches when using them on two of our evaluation scenarios. We have chosen rendering-based CL methods because they have the advantage that they are easily computable using the rendered depth buffer from the previous rendering iteration. This information is usually available for free in most rendering systems and in turn reduces

the CL computation overhead. In addition, we are interested in maintaining a high visual quality which implies that we have to include visual feedback information in the CL determination process.

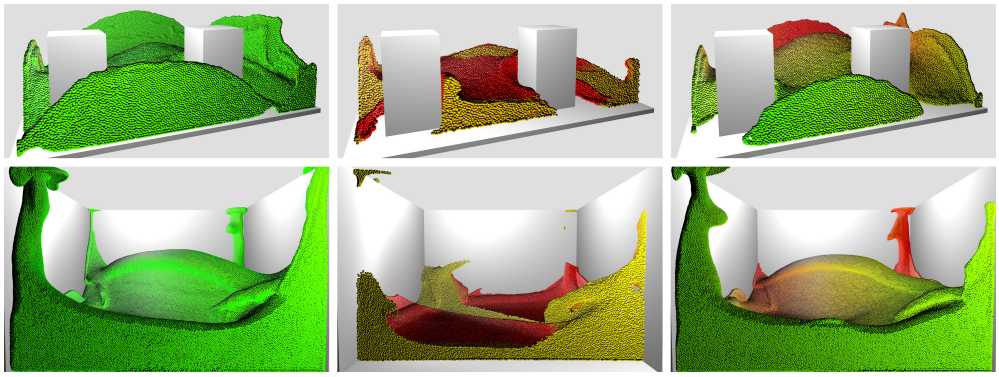

Figure 5.9: Both rows present sample CL visualizations of two evaluation scenarios to get an impression about the differences between different CLs. APBF models from left to right: High-CL particles classified by DTVS, low-CL particles classified by DTVS (without the high-CL particles), DTC classification (extended version of the image published in [KK16]).

5.1.3 Algorithm & Implementation Details

Algorithm 5 shows the APBF algorithm while highlighting differences to the original PBF algorithm from Macklin and Müller [MM13]. The parts that have been changed compared to the original PBF algorithm are highlighted in red. Green parts mark extensions to the PBF algorithm that have been added to the algorithm. Like in PBF, we apply external forces ($\mathbf{f}_{\text{ext}}$, e.g., gravity) that apply to all particles and predict all positions using the current velocity. In the next step, we detect all neighbors using the defined smoothing radius h for the simulation and determine all contacts based on the values of the signed distance field. At this point we apply an (optional) user-defined number of pre-stabilization steps that are included to avoid visual artifacts. The main solver loop itself is adapted to perform operations on active particles from the set $\hat{P}_l$ (see Equation (5.1)) only. This ensures that we skip inactive particles that have already reached their target position. Finally, we update all velocities and positions using the target positions x_i.

We implemented our APBF algorithm in C++ AMP [GM12] and realized collision detection with the environment with the help of signed distance fields (like in [Mac+14]). Particles were reordered in each simulation step to considerably improve the runtime performance of the neighboring particle discovery (due to improved coherence during memory accesses). We followed the counting-sort based approach by Hoetzlein [Hoe14] that uses efficient atomic

functions to reorder all particles. The evaluation of the CL assignments was implemented using depth information retrieved from the rendering module. This module rendered all particles as splatted spheres that were created using the geometry and pixel shaders, similar to the work by Laan et al. [LGS09]. After computing the depth image, we invoked a propagation kernel that computed CL information for all particles.

Algorithm 5: Simulation Loop based on PBF [KK16]

Input: CL information per particle to compute the sets $\hat{P}_l$

/* Apply forces and predict positions */

foreach *particle* $i \in P_1$ **do**
  $\mathbf{v}_i := \mathbf{v}_i + \Delta t \mathbf{f}_{\text{ext}}(\mathbf{p}_i)$;
  $\mathbf{p}_i^\star := \mathbf{p}_i + \Delta t \mathbf{v}_i$;
end
foreach *particle* $i \in P_1$ **do**
  Find neighboring particles $N(\mathbf{p}_i^\star, h)$;
  Find contacts for pre-stabilization;
end
while *iter* $<$ *stabilizationIterations* **do**
  Perform contact responses for contact C_i with $C_i \in \{C \mid \textit{particles}(C) \in \hat{P}_S\}$;
end
while *iter* $< I_l$ **do**
  foreach *particle* $i \in P_{iter}$ **do**
    Compute λ_i;
  end
  foreach *particle* $i \in P_{iter}$ **do**
    Compute $\Delta \mathbf{p}_i$;
    Perform contact responses;
  end
  /* Update future particle positions */
  foreach *particle* $i \in P_{iter}$ **do**
    $\mathbf{p}_i^\star := \mathbf{p}_i^\star + \Delta \mathbf{p}_i$;
  end
end
/* Update velocities and positions */
foreach *particle* $i \in P_1$ **do**
  $\mathbf{v}_i := \frac{1}{\Delta t}(\mathbf{p}_i^\star - \mathbf{p}_i)$;
  $\mathbf{p}_i := \mathbf{p}_i^\star$;
end

5.1.4 Visual Evaluation

Since APBF relies on choosing the number of solver iterations adaptively, deviations in terms of the actual simulation results cannot be avoided. Consequently, we analyzed the visual quality of APBF by visually comparing all results to a PBF-only version in which all particles have the highest possible CL. To compare the two simulation methods, we selected several frames and use semi-transparent particles to have the ability to compare both of them on a visual basis. Note that we do not need any pre-stabilization steps in the scope of the evaluation since visual artifacts do not appear in these cases. This is caused by the fact that we use high-resolution signed distance fields and small time steps.

We evaluated three scenarios using different fluid configurations for each of them: A water-like, an oil-like, and a jelly-like fluid. The solver was set up to use a fixed frame time of 16ms using two sub-steps with 8ms steps each. Each scenario was pre-configured with a minimum and a default number of iterations. The minimum number of iterations lead to an acceptable result that did not suffer from instability but did not yield the best visual results, which could be determined using the default number of iterations[1]. Both numbers have further been determined by analyzing the density distribution across all particles in the simulated volume. The configuration was then called *appealing* if the average density corresponded to the desired density of fluid in more than 90% of the cases. All evaluation scenarios used variations of widely used dam-break scenarios in which "block like" volumes of fluids was released at the beginning of the simulations [KK16]. Detailed information about the fluid setups are shown in Figure 5.10, which contains the number of particles and all solver iterations.

Scenario	Fluid	|Particles|	I min.	I max.	I adap.
Dam Break	Water	$\approx 216k$	3	6	$\{3, \ldots, 6\}$
Double-Dam Break	Oil	$\approx 673k$	5	10	$\{5, \ldots, 10\}$
Multi-Dam Break	Jelly	$\approx 225k$	4	8	$\{4, \ldots, 8\}$

Figure 5.10: Evaluation scenario configurations.

Figure 5.11 visualizes the first evaluation scenario while highlighting the differences between the high-quality and low-quality PBF versions in comparison to the APBF DTC and DTVS versions. The low-quality PBF simulation suffered from loosely connected particles with respect to their immediate neighbors, which lead to visible gaps in the visualized fluid surface. In contrast to this visualization, the high-quality PBF version did not suffer from these visible issue and had a smooth fluid surface. APBF with DTVS achieved the

[1]Note that increasing the solver iterations leads to an increase in terms of viscosity that is an inherit property of PBF simulations.

most appealing results which came closest to the fluid visualization characteristics (e.g., no visible holes in the surface) compared to PBF and preserved the fundamental properties of the flow like the vortex core in the front. However, using APBF with DTC resulted in visible artifacts in the posterior part of the simulation caused by particles with a low CL.

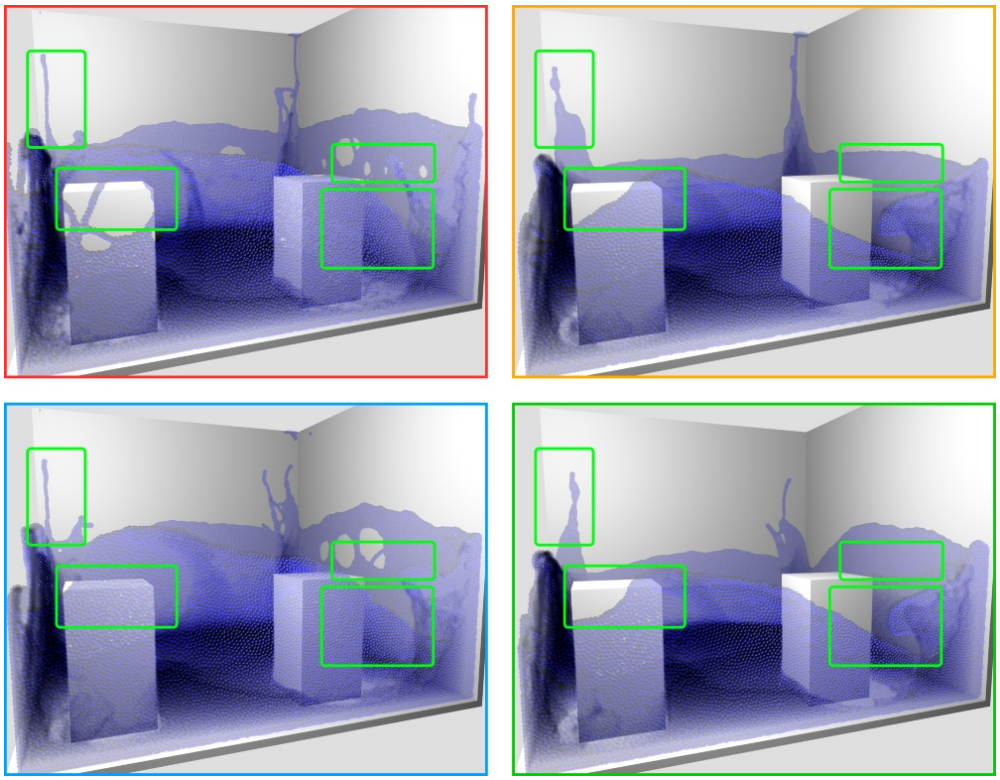

Figure 5.11: Evaluation image comparison for the dam-break scenario using a water-like fluid [KK16]. Color coding: Red: PBF with low quality ($I = 3$). Yellow: PBF with high quality ($I = 6$). Blue and green: APBF using $I \in \{3, \ldots, 6\}$), DTC left and DTVS right.

Similar results were visible in the second evaluation scenario (see Figure 5.12) which used high density and inviscid fluid. Two initially spawned fluid volumes collided in the center of the simulation domain with $\approx 336k$ particles each. The low-quality PBF versions showed an coarsely approximated fluid volume which was caused by the low number of iterations in the presence of a large number of particles. Further, holes in the surface were clearly visible which break the illusion of a smooth oil-like fluid with a reasonable surface tension. This particularly affected the central fluid-intersection area in the middle of the image and the fluid regions at the boundaries of the simulation domain. APBF (using either DTC or DTVS) ensured a high-quality simulation compared to PBF: It was able to preserve the fluid characteristics with respect to surface tension in general. However, DTVS achieved the best visual results and came closest to the reference simulation.

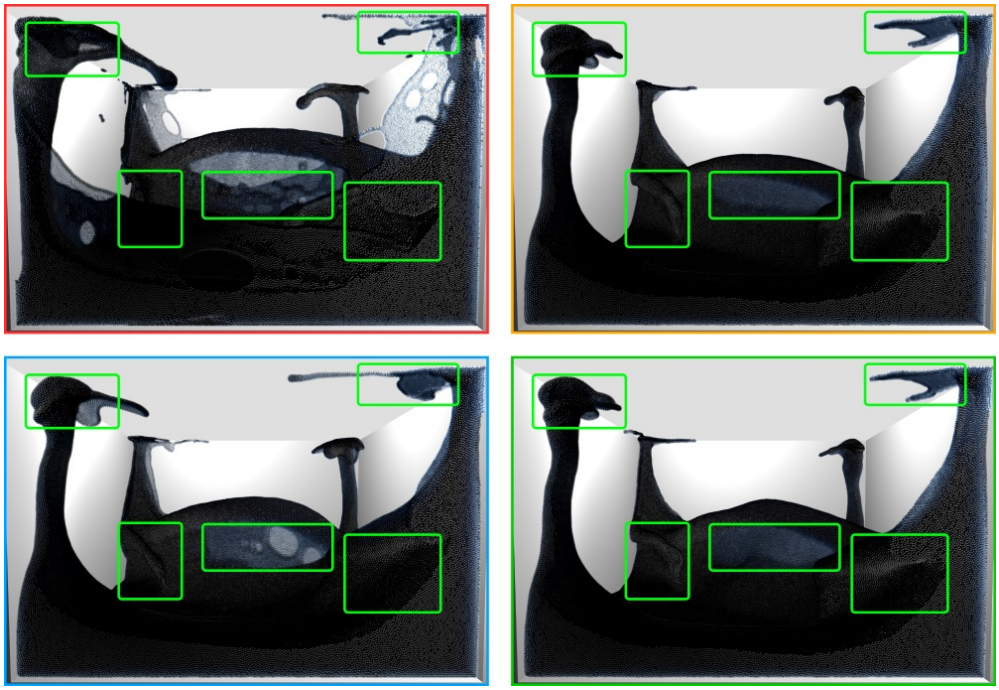

Figure 5.12: Evaluation image comparison for the double-dam-break scenario using an oil-like fluid [KK16]. Color coding: Red: PBF with low quality ($I = 5$). Yellow: PBF with high quality ($I = 10$). Blue and green: APBF using $I \in \{5, \ldots, 10\}$), DTC left, DTVS right.

The third scenario used a medium density, high viscosity fluid based on 4 fluid volumes overlapping in several regions. Each volume contained $\approx 56k$ particles which summed up to the total number of $\approx 225k$ particles. In this particular case, the visual differences mainly affected the spikes caused by the colliding fluid volumes. The APBF versions were both able to achieve high quality results when comparing them the high-quality PBF version. In contrast to the low-quality PBF version, APBF performed significantly better by preserving more fluid characteristics. Again, DTVS performed slightly better as a higher classification was assigned to all visible particles at the same time.

Figure 5.14 depicts the influence of APBF and PBF on the average density of the fluid simulation during the performed 1000 steps of each evaluation simulation. As shown in the different density deviation graphs, APBF had a most significant deviation of 4% (measured in the first scenario) compared to the reference PBF simulation in all cases. All other deviations were around 1.5% which can be considered negligible with respect to the performance improvements and the high visual quality of the simulation. The APBF graphs are just slightly shifted and scaled in comparison to the reference PBF graph. Depending on the scenario, DTVS performed slightly worse compared to DTC although it looked visually more similar to the high-quality reference simulation. This is due to the fact that the visible parts received a higher CL which in

turn caused the particle positions to be adjusted more precisely. In sum, APBF overall achieved competitive average densities with respect to PBF, which also explains the small visualization variations analyzed above.

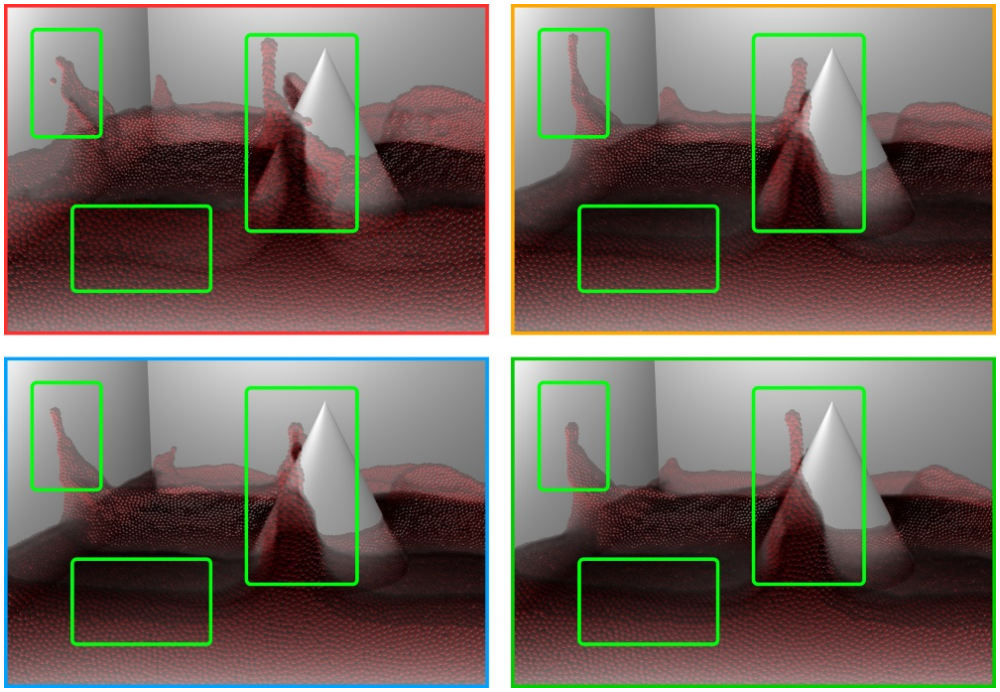

Figure 5.13: Evaluation image comparison for the multi-dam-break scenario using a jelly-like fluid [KK16]. Color coding: Red: PBF with low quality ($I = 4$). Yellow: PBF with high quality ($I = 8$). Blue and green: APBF using $I \in \{4, \ldots, 8\}$), DTC left and DTVS right.

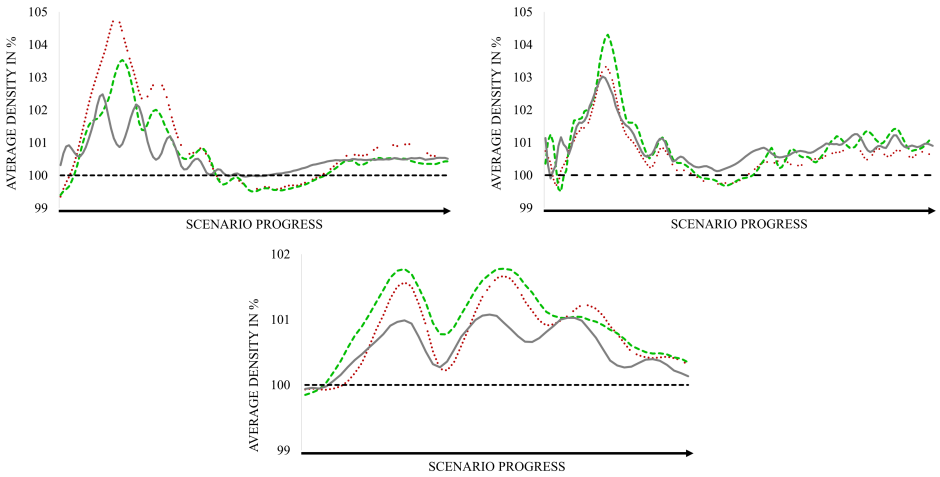

Figure 5.14: Average density deviations for all evaluation scenarios (reference PBF in gray) [KK16]. Rest density is highlighted by the black dotted line. Red: APBF using DTC. Green: APBF using DTVS.

5.1.5 Performance Evaluation

In order to evaluate the performance of APBF, we gathered performance measurements on two different GPUs featuring different architectures: AMD Radeon HD 7850 and NVIDIA GTX 680 [KK16][2]. We then determined the actual speedup compared to PBF to demonstrate the performance gains possible with our method.

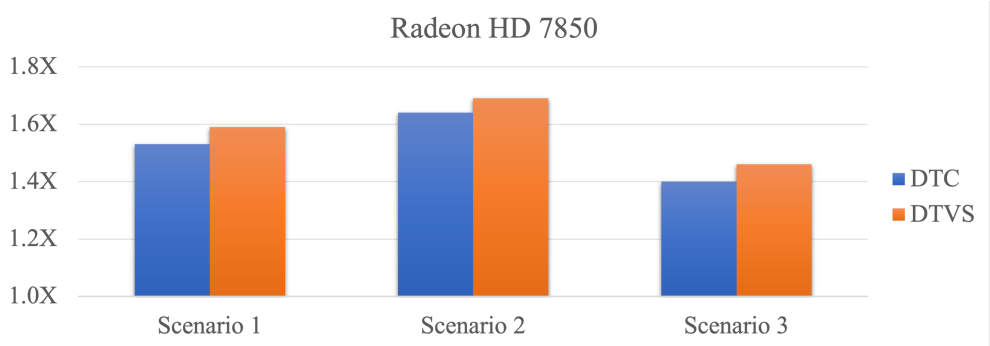

Figure 5.15: Speedup of APBF compared to PBF on Radeon HD 7850 (higher is better, based on performance numbers from [KK16]). Color coding: Blue: DTC; Orange: DTVS.

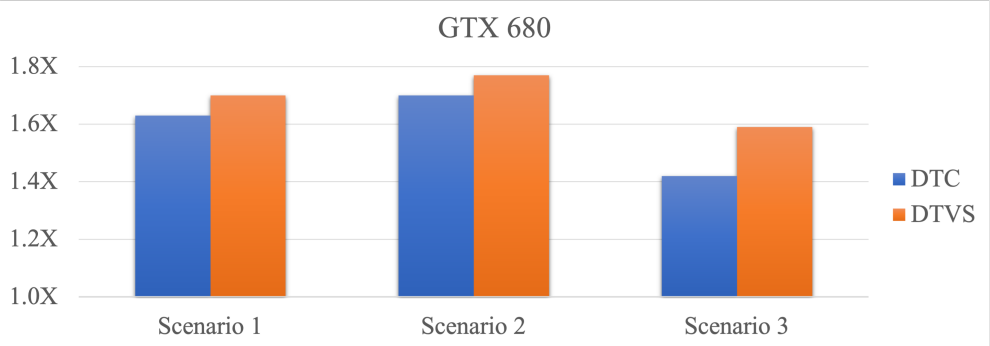

Figure 5.16: Speedup of APBF compared to PBF on GTX 680 (higher is better, based on performance numbers from [KK16]). Color coding: Blue: DTC; Orange: DTVS.

In the scope of the evaluation, a single performance measurement was determined by computing the median execution steps across 100 applications runs and 1000 simulation steps. This compensated different OS/driver imposed overheads and slight deviations between different frames. The overall frame time was about $\approx$ 30ms in scenario 1, $\approx$ 290ms in scenario 2, and $\approx 50ms$ in

[2]Note that both GPUs were state-of-the-art at the time of the publication. Therefore, we retained these measurements to present the original results. However, it is of interest to reevaluate our method on more recent GPUs in future work (see Chapter 14).

scenario 3 using our APBF approach. Figures 5.15 and 5.16 show the speedups comparing our APBF method to the base line PBF model on all three evaluation scenarios using our DTC and DTVS adaptation models (see Section 5.1.2).

Both GPUs behaved similarly regarding their overall runtime behavior on all scenarios, in which the NVIDIA GPU performs slightly faster on all benchmarks. We measured speedups of 1.5× to 1.7× compared to the base line PBF model in the first two scenarios. Scenario 3 yielded considerably smaller speedup in total since both, camera and all particles are closer to the fluid volume/the visible surface resulting in less potential to optimize.

In all cases, our distance-to-the-visible-surface adaptation method (DTVS) yielded slightly better results. It improved the speedup by additional $\approx 0.1\times$ on both GPUs for scenarios 1 and 2. In scenario 3, the NVIDIA GPU benefits considerably more from the reduced number of particle adjustments and DTVS improved our measured speedup by additional $\approx 0.2\times$.

5.2 Screen Space Particle Selection

Inspired by previous work in fluid simulations, this section covers another challenging task: Given a certain *snapshot* of a simulation state, researchers often seek to gain deeper insights into the dataset [Tuk77]. Data exploration using 3D visualization techniques is well-suited for this purpose. In addition, there have been a number of different approaches in recent years that allow fine-grained interaction with datasets. Most prominent in this domain are 3D selection techniques that are inherit challenging to realize due to the high number of (often unstructured) data elements to process and the available degrees of freedom. Methods going further than the visual UI level are based on 3D selection volumes, which are challenging to realize efficiently [ONI05]. Efficiency here refers to runtime performance and memory consumption as well as the actual interaction possibilities exposed to the user, the scientists in this context (see Figure 5.17).

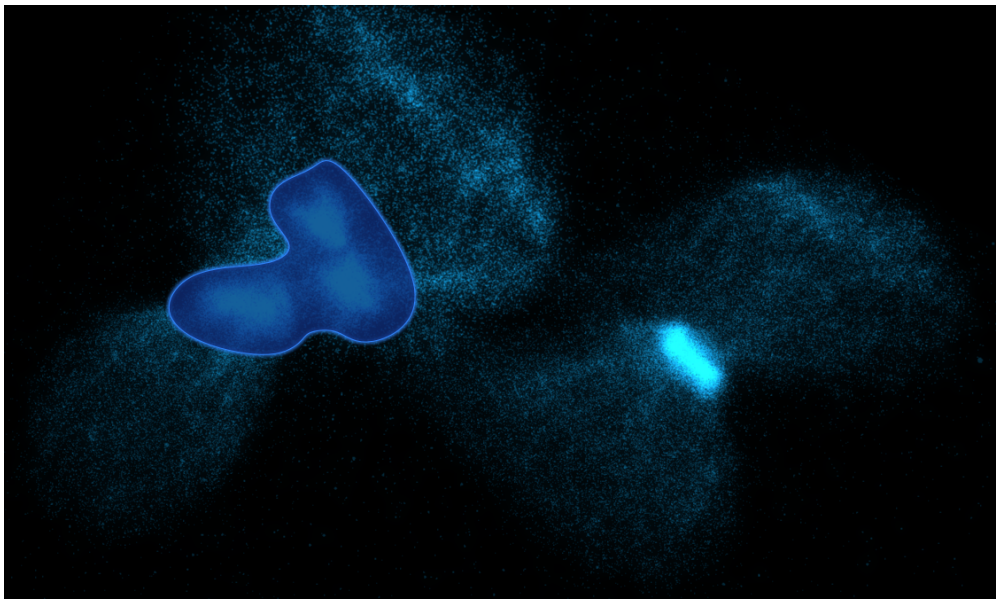

Figure 5.17: Triple-cluster selection in the context of a large particle gravity-simulation dataset.

Each *data point* (also referred to as a *particle* in this context) is associated with certain scalar properties (like its physical mass in space). A subset of them is then usually used during the selection process to either build the selection volume or guide the selection algorithm [ONI05; Yu+12; KK18]. Two state-of-the-art methods are named *CAST* algorithm family, which is an abbreviation for context-aware selection techniques [Yu+12; Yu+16]. They operate on selection lassos in 2D that are drawn by a user. This data is then used to create a selection volume by analyzing the contour and overall shape of the user's lasso. As outlined in Chapter 4, the CAST family relies on a local density estimator

combined with the *Marching Cubes Algorithm* MCC [LC87]. MCC relies on 3D grid in memory and is applied to the dataset to extract an isosurface, which is then used to distinguish between selected and unselected particles. Due to the high processing complexity of the CAST methods, large datasets can only be processed efficiently if the accuracy of the selection method is reduced; mainly by reducing the size of the 3D grid(s) for the MCC algorithm and for the calculation of the local density estimator. This substantially limits their applicability to real-world datasets.

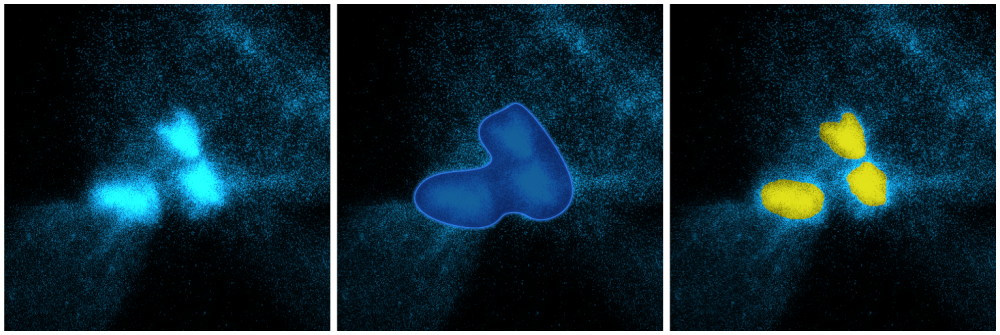

Figure 5.18: A visualized selection process in SSPS from left to right [KK18]. The user starts by drawing a selection lasso (middle), releases the mouse and the selected set of particles is visually highlighted for further processing (right). This task takes up to several minutes using related approaches. SSPS requires only a few milliseconds to complete the selection process, consuming one hundredth of the memory compared to CAST.

This section presents our *Screen Space Particle Selection* (SSPS) method that overcomes these limitations by reducing the computational complexity and drastically decreasing the required amount of memory. Our method is specially designed for GPU architectures and other massively parallel processing units. The proposed approach has a runtime complexity of only $O(n \cdot k)$, where n is the number of particles in the scene and k is the number of average neighbors per particle [KK18]. This enables us to reduce the execution time to several milliseconds, even in the scope of datasets containing millions of particles (see Figure 5.18). Similar to CAST, we also use the basic input via a 2D lasso-based selection shape drawn by the user. Unlike other work in this field and to the best of our knowledge, we are the first leveraging the SPH concept to perform actual particle selection operations. Using SPH allows us to benefit from well known local density estimation methods [Luc77; Mon92] that can be efficiently computed using specially designed neighborhood exploration methods [Gre10; Hoe14; GKK19]. It also removes the need for an explicitly materialized global 3D density grid, to which the CAST methods apply the MCC algorithm to. Instead, our idea is to use 2D screen-space buffers that can be extracted from the rendering system and perform our selection analyses mainly in screen space.

This decouples the selection analyses from the underlying complexity of the input dataset, which in turn are required to distinguish between selected and unselected particles. Decoupling in combination with GPU acceleration enables us to finish a particle selection task in the scope of milliseconds (also on real-world datasets).

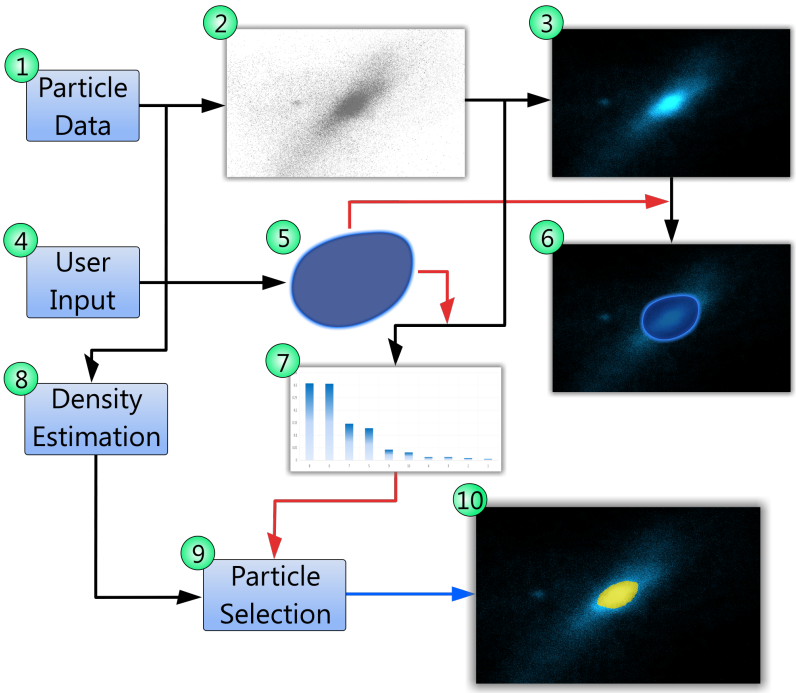

Figure 5.19: The SSPS processing pipeline. (1) Particle data is rendered to build a depth image (2) and the actual visualization of the dataset (3). The user performs an input operation (4) by drawing a selection lasso (5) which is then added on-the-fly to the rendered image in order to give visual feedback (6). Also based on the selection lasso, we derive the mask image (7) that is combined with information from our density estimation pipeline (8) to determine all selected particles (9). Finally, the selection task is completed and the selected set of particles is visually highlighted and presented to the user (10, based on the workflow visualization published in [KK18]).

Figure 5.19 shows our SSPS processing pipeline consisting of ten steps in total. As outlined above, most of the analysis to map a user's selection lasso to the actually particles to be selected happens in screen space. The initial visualization pipeline allows us to retrieve the depth image for further analysis in the 2D space (steps $1-3$). Note that the depth buffer contains linearized depth information within the interval $[0.0, \ldots 1.0]$, where 0.0 refers to the near plane and 1.0 to the far plane. A user draws a selection lasso spanning the

potentially interesting subregions of the rendered volume (step $4-6$). Afterwards, we use the information that is implicitly contained in the shape of the lasso (step 7) to build an *intention buffer* (IB, see Section 5.2.1). This buffer bridges the gap between the 2D screen space and the 3D dataset, which contains per-pixel information about the mask in the `fp32` format. The basic idea is that the IB maps the selection mask from the screen space to 2D plane in space that represents a slice through the dataset. Afterwards, we can use a particle density estimation analysis (step 8) which provides us with crucial information about the local density around each particle. The 2D slice from the previous step will be extended to a 2D box in space that allows us to compute a set of initial particles based on their densities that reflect the user's intention best (based on the shape of the lasso). This information is then combined in step 9 with the information from the IB, which uses a fast and efficient flood-filling-like algorithm to transitively select particles which match our selection criteria (specified by the IB).

5.2.1 Lasso Selection

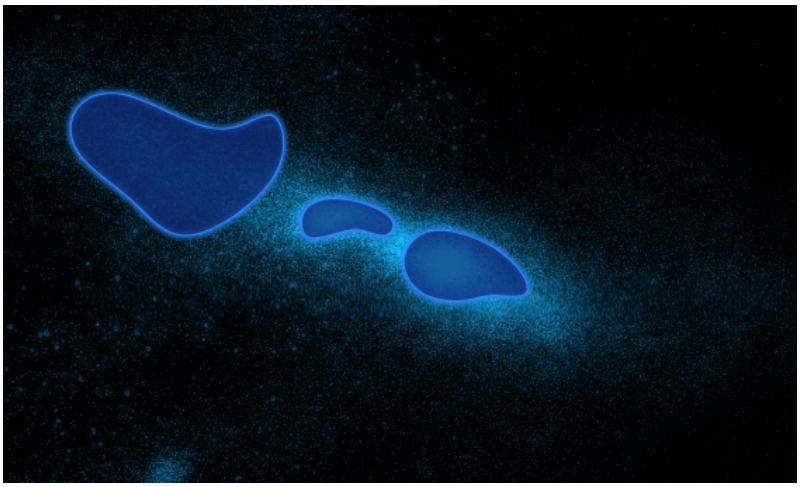

Figure 5.20: A multi-cluster dataset visualization with three smoothed selection lassos [KK18].

We start by computing the *selection mask* from a selection lasso. This lasso is given by the user in the form of a polygon defined by a collection of 2D points in screen-space coordinates. The input polygon, will be automatically closed by connecting the last point with the first point if it not closed properly by the user (also a common technique used by many related methods [Yu+16; KK18]). We optionally apply an additional Laplacian smoothing [VMM99] step to denoise the overall shape of the selection lasso. This particularly helps to remove sharp edges of the polygon which in turn ensures a better mapping to the desired 3D selection (see Figure 5.20). Better here means more resistant to outliers, which can be introduced either by the user input itself via the

input device used or by small deviations due to numerical inaccuracy in the conversion between coordinate spaces.

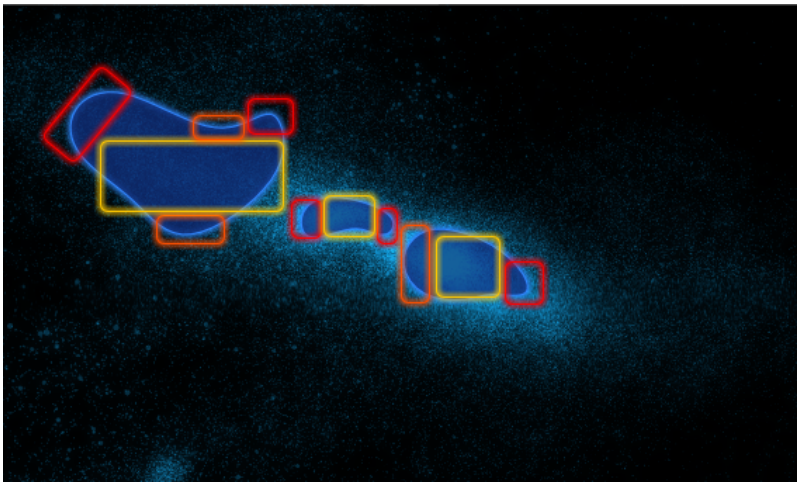

Figure 5.21: Figure 5.20 with annotated regions of interest in the context of three selection lassos [KK18]. To distinguish between more important and less important regions of interest, we can use the implicit information of the shapes of the lassos. Color coding: Yellow: Areas which should be included in the selection, as they are close to the centroid of the lasso. Orange: More important areas compared to the yellow ones, as they represent the separation of desired and undesired areas. Red: The most important areas, as they are farthest from the centroid and have more curvature compared to other areas. This indicates that a user wants to clearly separate between included and excluded particles.

As mentioned in the introduction to SSPS, there is a lot of implicitly given information that can be derived from the selection lasso. Figure 5.21 shows highlighted regions of interest from which we can gather additional insights regarding a user's intended selection. For example, a lasso contains sharp corners to separate regions to be selected from regions that should not be included. Similarly, large areas that span a decent number of pixels indicate areas in projected 2D space that should be included in the final selection. We have evaluated different approaches to infer this kind of information from the selection lasso. In turns out that applying a weighting kernel to all pixels within in the lasso works excellently on our evaluation scenarios (see Section 5.2.4). In order to perform further operations on the input selection lasso, we first render it to a 2D image (referred to as the *mask image*). Afterwards, we apply the mentioned position-dependent weighting kernel to the mask image, This results in an *importance factor* $\in [0.0, \ldots, 1.0]$ for each pixel representing how important a specific pixel is for the current selection process (a higher value indicates more importance). Note that while we could also use the selection

polygon itself, it is more convenient for us to work with this rasterized mask representation of the selection lasso in the following steps.

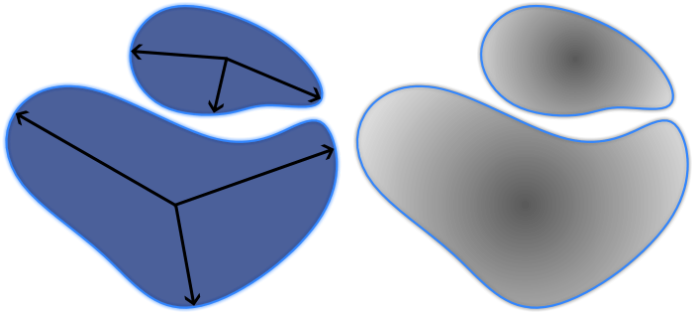

Figure 5.22: Two selection lassos with different shapes (left) and their rendered mask images (right) [KK18]. The more important a pixel in the mask image is, the higher its value will be. This is encoded in the image via brightness levels, where a white pixel refers to a high weight and a black pixel refers to a low weight. In order to compute these values, we use a radial kernel that weights each pixel based on its distance to the centroid of the polygon (visualized using black arrows, left).

We use a radial weighting kernel (see Figure 5.22) that allows us to preserve major characteristics of the selection lasso [KK18]

$$W_M : \mathbb{N} \times \mathbb{N} \mapsto [0.0, \ldots, 1.0], \tag{5.8}$$

where the first two arguments of W_M are the x and y coordinates of the current pixel. The distance between each pixel within the lasso and the centroid of the polygon is used to determine its weight. The farther away a pixel is, the more important it is:

$$W_M(x, y) = \frac{||(x, y) - C_P||}{d_{\max}}, \tag{5.9}$$

where C_P refers to the centroid of the mask polygon and $d_{\max}$ is the maximum distance between two arbitrary points of the polygon

$$d_{\max} = \max_{p_i, p_j} ||p_i - p_j||. \tag{5.10}$$

A straight-forward algorithm to implement mask construction is shown in Algorithm 6 that accepts a list of polygon points and constructs the actual mask image M_I. This pseudo code is straight forward to implement in a desired programming language. However, implementing it using a GPU kernel to apply the weighting kernel W_M to each pixel greatly speeds up the processing speed within the selection pipeline.

Algorithm 6: Straight-Forward Mask Construction Algorithm

Input: implicitly-connected lasso points P_L
P := Construct 2D polygon from P_L;
P_S := Smooth 2D polygon P;
Compute centroid C_P of the polygon;
Compute $d_{\max}$ according to Equation (5.10);
M_I := Construct mask image initialized with 0;
foreach *mask pixel* (x, y) **do**
  if (x, y) *is inside selection polygon* P_S **then**
    $M_I(x, y)$:= Compute W_M according to Equation (5.9);
  end
end
return M_I

Based on the information extracted from the mask image M_I, we build the intention buffer for further processing steps. In this scope, the intention buffer I_B is an intermediate data structure that allows us to map the contents of M_I to a 3D slice in space through the dataset The underlying algorithm uses a histogram distribution of all depth values. The histogram itself is constructed using a given set of bins to which the depth values will be assigned and added to. The fewer bins we use, the less strict the selection becomes in the end on the one hand. On the other hand, the more bins we use, the finer-grained the selection becomes, as it pays more attention to the separation into individual bins based on the mask image M_I. However, the amount of histogram bins SI is a user-defined variable that depends on the scenario and user preferences in general. We set this number to 16 for all experiments and found out that this number essentially resulted in the highest precision on all use evaluation scenarios (see Section 5.2.4).

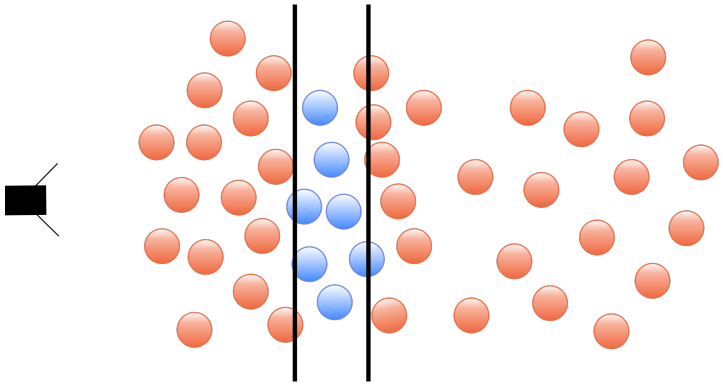

Figure 5.23: A volume slice through a dataset defined by two planes.

The 3D slice through the volume is built by computing a near and a far plane to focus on particles that fall into this slice (see Figure 5.23). To determine both planes, we use the depth-distribution values stored inside the intention buffer. Figure 5.24 shows a visualized dataset and its corresponding linearized depth image. The high-level idea is to use the input mask image and weight each depth value by its corresponding mask weight in the 2D mask image M_I. Hence, the selection intention given by M_I can be mapped via the analogy that a high-weight value in the mask image means a high intention to include a particular pixel in the selection process. Figure 5.25 visualizes the difference between weighted and non-weighted (raw) depth values. As can be seen in this comparison plot, M_I significantly influences the distribution stored in the intention buffer, which in turn implicitly influences the determination of the near and far planes of the desired 3D volume slice.

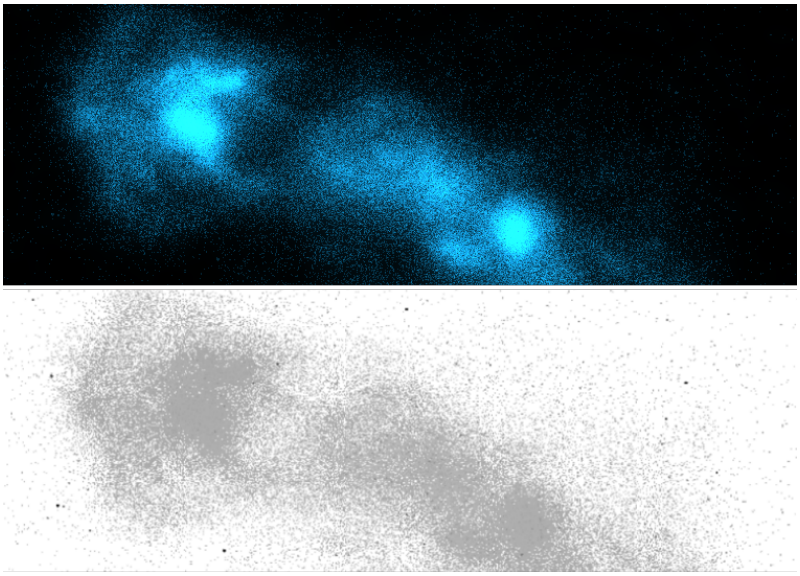

Figure 5.24: A sample visualization of a dataset (top) and the corresponding linearized depth image (bottom) [KK18].

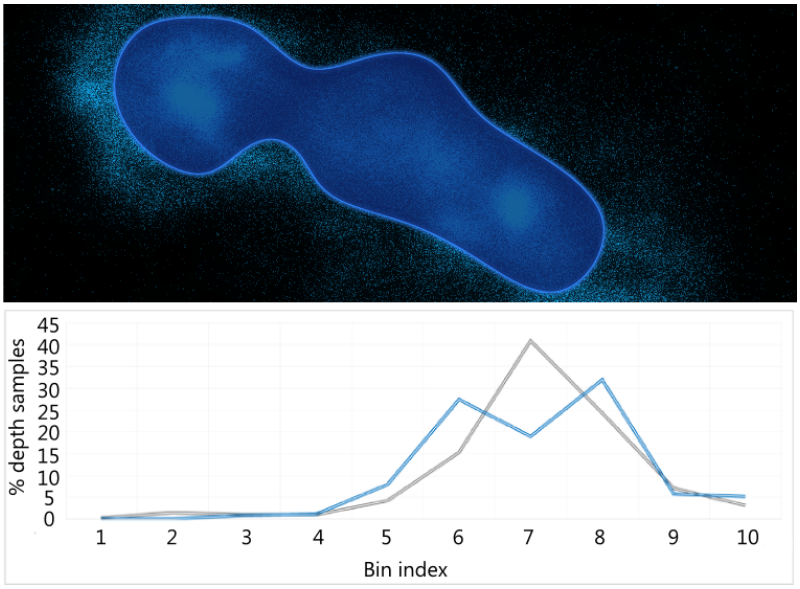

Figure 5.25: Figure 5.24 with a complex selection mask spanning multiple clusters (top) [KK18]. The X-axis refers to the source index of the original bin, whereas the Y-axis represents the % of depth samples in all bins.

Figure 5.25 also indicates that a greedy solution to take the bin with the largest number of accumulated weights for the volume slice might is not the perfect solution. In this case, bins 8 and 6 contain the most weights depending on the depth image and the shape of the mask. However, both the selection mask and the computed mask image M_I are not perfect. This means that the accuracy of the lasso and all weights are primarily an approximation of the user's intent. Mapping this approximation to a greedy solution by taking only one bin in all cases into account can easily lead to unacceptable imprecision. In the scope of Figure 5.25, this would result in bin 8 being considered and bin 6 being neglected, even though these are the two most significant bins containing the largest number of depth samples. Figure 5.26 shows the corresponding bins from Figure 5.25 sorted by the % of depth samples within each bin. In numbers, the difference between bin 8 and bin 6 in this sample is about 7%, while the deviation to the bin 7 is greater than 10%.

For the actual determination of the 3D slice, which concludes the first two processing steps of the user input (steps 5 and 6 in Figure 5.19), we first sort all bins in descending order. Next, we make a single pass over all bins from left to right (see Figure 5.26, the bin with the most samples (left) to the bin with the least samples (right)). Then, we compute the differences between the current bin and the next bin (it's successor bin). We include all bins in the iteration as long as the calculated difference is below the average difference computed using all bins. This ability considerably reduces the tendency of our method to favor selection ranges with similar weights.

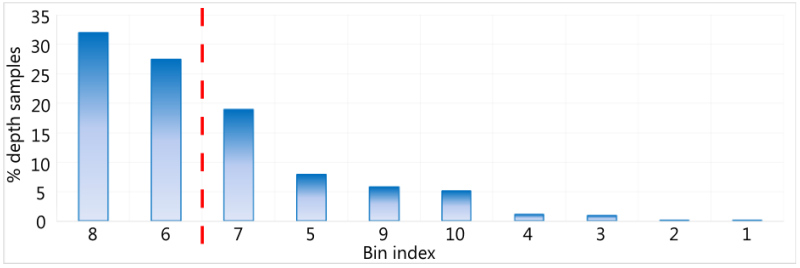

Figure 5.26: Intention buffer bins from Figure 5.25 [KK18] that have been sorted according the maximum number of depth samples in %. The X-axis refers to the source index of the original bin, whereas the Y-axis represents the % of depth samples in the bins. The red dashed line indicates that our approach uses the first two bins in this sample, rather than considering only bin 8.

Algorithm 7: Our 3D Volume Slice Algorithm [KK18]

Input: mask image M_I, linearized depth image D, intention buffer I_B

`/* Compute min and max depth values from the area affected by the mask image */`

$D_{\text{min}} := 1$, $D_{\text{max}} := 0$;
foreach *pixels* (x, y) **where** $M_I(x, y) > 0 \wedge D(x, y) < 1$ **do**
    $D_{\text{min}} := \min(D_{\text{min}}, D(x, y))$;
    $D_{\text{max}} := \max(D_{\text{max}}, D(x, y))$;
if $D_{max} < D_{min}$ **then**
    return No slice found;

`/* Compute sampling interval` D_S `*/`

$D_S := \frac{D_{\text{max}} - D_{\text{min}}}{\text{number of samples}}$;
Initialize I_B with 0;

`/* Compute intention buffer buckets` I_B `*/`

foreach *pixel* (x, y) **where** $M_I(x, y) > 0 \wedge D(x, y) < 1$ **do**
    $i_b := \frac{D(x,y)}{D_S} \cdot$ number of bins;
    $I_B(i_b) := I_B(i_b) + M_I(x, y)$;

`/* Determine itention-buffer based near and far cut-off planes */`

Sort I_B in descending order of ratings;
$d_I :=$ average difference of $I_B(i)$ and $I_B(i+1)$ in I_B;
$i := 0$;
while $i <$ *number of bins* $- 1 \wedge I_B(i) - I_B(i+1) < d_I$ **do**
    $i := i + 1$;

`/* Return a pair containing the near and far planes */`

return $\text{depth}(I_B(0)), \text{depth}(I_B(i))$;

Algorithm 7 represents a ready-to-use pseudo-code algorithm to construct the 3D volume slice for further processing. It uses a mask image M_I, a linearized depth image D from the rendering system, and an allocated intention buffer I_B to read from and write to. To access the depth image, we use the notation $D(x, y) \mapsto [0, \ldots, 1]$, where 1 refers to the far and 0 refers to the near depth plane in the visualization pipeline. Similarly, we access the mask image M_I via $M_I(x, y) \mapsto [0, \ldots, 1]$, where 1 refers to a high weight from our weighting kernel (see Equation (5.9)). First, we iterate over all pixels in 2D screen space. Afterwards, we test whether they are affected by the mask image and contain a valid depth sample ($D(x, y) < 1$, lines 1–6). If we are not able to find a valid depth interval ($D_{\max} < D_{\min}$), there are no particles contained in the depth image in presence of the selection mask. If we detect a valid depth interval, we compute the internally used sampling interval D_S, to map depth values to their appropriate bin inside of I_B (line 7).

After initialization of I_B to 0 (a *memset operation*), we fill to intention buffer data structure by accumulating mask weights from M_I (lines 8–11). Note that I_B will not contain any information from pixels which are neither covered by M_I nor contain a particle in D. We sort all bins (as depicted in Figure 5.26) and compute the number of bins that we want to include in the selection process (lines 12–16). Finally, we map the actual depth values from the screen-space depth buffer of the selected bins from I_B (line 17).

5.2.2 Density Estimation and Particle Selection

The actual particle selection step (see Figure 5.19) involves a deep volume analysis. We follow approaches from related works that use density estimation methods to perform the actual selection of particles within the volume. As mentioned in the beginning and in contrast to related methods, we use an efficient SPH-based density estimator [GM77; Luc77; Mon92] (see also Equation (4.1) and Chapter 4)). Instead of approximating an arbitrary quantity A_i we want to approximate the density itself via [KK18]

$$\rho_i = \sum_j \frac{m_j}{\rho_j} \rho_i W(||p_i - p_j||, h) = \sum_j m_j W(||p_i - p_j||, h), \tag{5.11}$$

where m_j is the mass of each particle and h the smoothing length for our weighting kernel W. Since we want to focus on the spacial density that should not contain any particle-specific mass information, we can further rewrite Equation (5.11) to [KK18]

$$\rho_i = \sum_j m_j W(||p_i - p_j||, h) = \sum_j W(||p_i - p_j||, h). \tag{5.12}$$

However, these approximations work out quite well if the underlying particle distribution is more or less uniform. Otherwise, some particles have more neighbors that are within the smoothing radius h than others (see Figure 5.27).

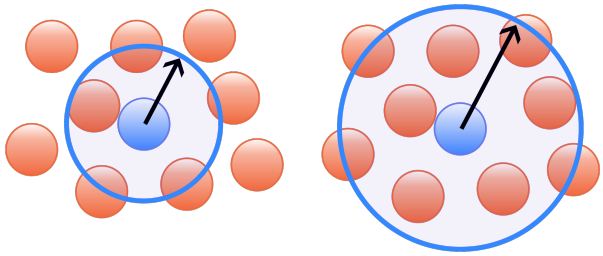

Figure 5.27: Different smoothing radii for our density computation that result in a different number of neighbors influencing the quality of an approximated smoothed quantity.

To ensure stable computation of SPH-based quantities, h is often adaptively chosen such that each particle has an appropriate number of particles with respect to the smoothing kernel W [NP94; KSW99]. This changes equation Equation (5.12) to

$$\rho_i = \sum_j W(||p_i - p_j||, h_i), \tag{5.13}$$

where h_i is a context dependent and particle-specific smoothing length [KK18]. Figure 5.28 visualizes the benefit of different smoothing radii in the presence of different densities in the dataset.

Unfortunately, we cannot determine an appropriate smoothing length before loading the dataset because we do not have information about the local particle

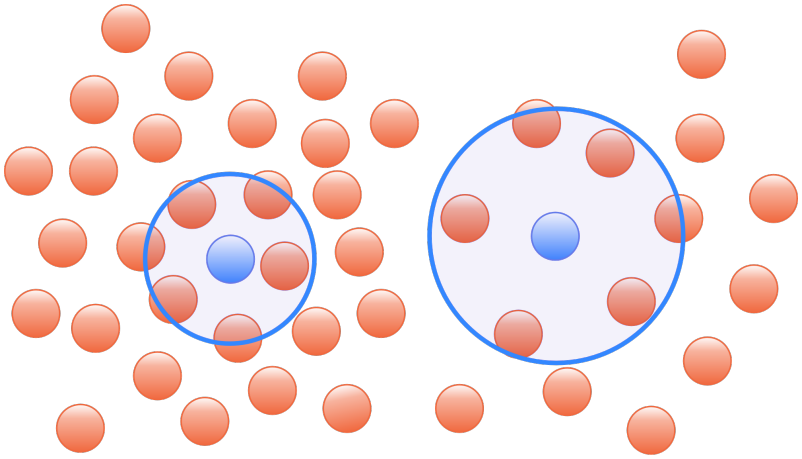

Figure 5.28: Adaptively chosen smoothing radii on the same dataset.

densities at that time. We again follow related publications that use on-the-fly methods to approximate the density in the dataset with adaptively chosen smoothing radii for each particle. The main task of such an algorithm is to find a high-quality estimation of h_i for each particle in the dataset. In the case of our SPH-based method, we make use of a correlation between the ideal number of neighbors N_h that is weighting-kernel dependent [WHK16]. Dehnen and Aly [DA12], for example, have shown important findings about the ideal number of neighbors in the presence of different weighting kernels. For example, a well-known kernel in the field of astrophysical simulations is a *cubic spline kernel* [KK18], which requires $N_h \approx 42$ neighbors per particle [DA12]. Our idea is to use the high-level concept of Winchenbach et al. [WHK16] based on the explained number of ideal numbers N_h. As presented by Winchenbach et al., we also leverage the *adaptive volume* formula by Solenthaler and Pajarola [SP08] to determine information about the spatial distribution of all particles in the dataset, which is given by

$$V_i = \frac{1}{\sum_j W(||p_i - p_j||, h_i)} = \frac{1}{\rho_i}, \tag{5.14}$$

where V_i is referred to as the inverse particle volume and is equivalent to the inverse of our particle density defined in Equation (5.13). Following Winchenbach et al. [WHK16], they relate V_j to the smoothing length h_i by

$$h_i = \lambda_s \cdot V_i^{\frac{1}{3}} \left(\frac{N_h}{\frac{4}{3}\pi} \right)^{\frac{1}{3}}, \tag{5.15}$$

where λ_s is a scalar global scaling factor. The underlying idea of their approach is to answer the question of how many particles with radius h_i intersect with a sphere based on V_j or neighboring particles j. Figure 5.29 visualizes this concept by presenting a smoothing radius h_i in the context of several spheres with radius V_i.

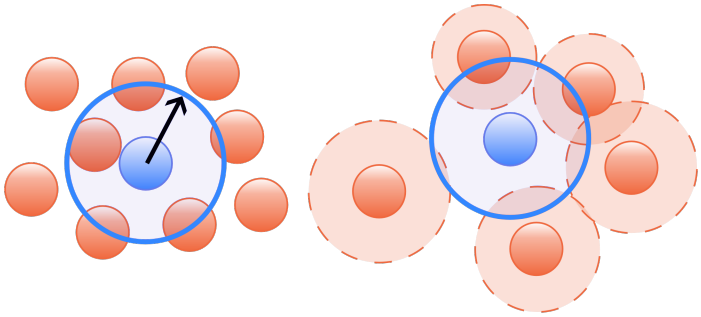

Figure 5.29: A smoothing radius h_i of a blue particle. Left: in the context of other particles. Right: in the context of different particle volumes V_j of its neighboring particles j (red circles).

To get a global approximation of all smoothing radii h_i, we use an iterative algorithm that starts by choosing an initial smoothing radius $h_{\text{init}} = \text{init}(h_i)$ for each particle. To obtain a *good initial value* for h_i (called h_a), we assume that all particles are distributed uniformly within the dataset. h_a is linked to the maximum spacial extent dataset via [KK18]

$$d_i = \frac{\dim_i(\text{dataset})}{\text{number of particles}}, \tag{5.16}$$

where $\dim_i$ is the ith dimension of the bounding box including all particles in the dataset. h_a is then given by

$$h_a = \max\left(d_x, d_y, d_z\right). \tag{5.17}$$

Having calculated a rough approximation for h_i, we consider the neighborhood of each particle and the weighting kernel W. For our first conjecture, we assume that a cubic increase in the number of neighboring particles can be achieved by a linear increase in radius h_i [KK18]

$$\text{init}(h_i) = \left(\frac{\sum_j W_N(||p_i - p_j||, h_a)}{N_h}\right)^{\frac{1}{3}} h_a = \left(\frac{\rho_i[h_i = h_a]}{N_h}\right)^{\frac{1}{3}} h_a, \tag{5.18}$$

where W_N is a weighting kernel applied each neighbor of the ith particle that is in radius h_a. All other particles with not be considered [KK18]

$$W_N(d, h) = \begin{cases} 1 & \text{if } d < h \\ 0 & \text{else.} \end{cases} \tag{5.19}$$

Note that this weighting kernel is symmetric and has finite support, which is a common requirement for SPH kernels [Mon92; MCG03]. In our case, the weighting kernel is not normalized, which is not required because we divide by the number of neighbors in the approximation algorithm. This results in the kernel being normalized at the end when being used within the iterative

algorithm. Although there may be edge cases where the normalization does not work quite smoothly in practice, the impact of these deviations is negligible since we interpolate h_i between multiple iterations. This compensates for small deviations with respect to the kernel normalization.

Algorithm 8: Our Density Estimation Algorithm [KK18]

Input: particle data, maximum number of iterations i_{max}

foreach *particle* i **do**
  $h_i := \text{init}(h_i)$ according to Equation (5.18);
end
$\Delta h := 0$;
$i := 0$;
do
  foreach *particle* i **do**
    Compute V_i according to Equation (5.14);
  end
  $\Delta \hat{h} := 0$;
  foreach *particle* i **do**
    Compute $\hat{h}_i$ according to Equation (5.15);
    `/* Update` h_i `by smoothly moving towards` $\hat{h}_i$ `[Mon92] */`
    $\Delta \hat{h} := \Delta \hat{h} + ||h_i - \hat{h}_i||$;
    $h_i := h_i + \frac{\hat{h}_i - h_i}{2}$;
  end
  $\Delta \hat{h} := \frac{\Delta \hat{h}}{\text{number of particles}}$;
  if $||\Delta h - \Delta \hat{h}|| < \epsilon \vee i \geq i_{max}$ **then**
    break;
  end
  $\Delta h := \Delta \hat{h}$;
  $i := i + 1$;
while;

Unlike other (iterative) density approximation methods, we do not dynamically adjust h_i over the runtime of the program, since we want to analyze a particular snapshot of the simulation data at the "selection time". Therefore, we compute a current approximation of h_i immediately when we perform a particle selection task. To do this, we apply several iterations until the average Δh_i between two steps falls below a certain threshold or the maximum number of iterations is reached. The upper bound on the number of iterations is necessary to ensure deterministic worst-cast execution time, which leads in turn to an increase in responsiveness. Note that $\text{init}(h_i)$ does not need to be recomputed each time, as we can safely compute this approximation when loading the dataset. If the dataset changes over time (instead of a static dataset), the initial value for h_i should be recomputed for each selection task to improve the convergence speed of our method.

Algorithm 8 gives a detailed pseudo-code description of our density estimation step. First, we determine the initial approximation for h_i for all particles (lines 1–3). As mentioned above, this step can be precomputed upon loading the dataset in the case of a non-changing dataset. Afterwards, we perform the iterative density estimation that does at least a single step (`do-while` loop). We calculate the particle volume V_i for all particles (lines 7–9) before adjusting the individual smoothing radii h_i by interpolation (lines 11–15). Here, the variable $\Delta\hat{h}$ (line 10) represents the average change over all particles in the dataset (lines 13 and 16), which is used to determine whether to break the loop. Alternatively, we also break when the maximum number of iterations has been exceeded, as described above (lines 17 and 21). If GPU kernels are used to compute every `foreach` loop (as in our implementation), the value of $\Delta\hat{h}$ needs to be computed using additional reduction operations/steps, which is not shown here.

Once we have computed the density information of each particle, we have to select the desired particles. For this purpose, we utilize a flood-filling algorithm. Flood-filling itself is based on the implicitly available particle-neighborhood relationship defined via the densities of all particles: We identify clusters based on particles that have similar densities in their surrounding. Using the available density information avoids the construction fo an explicit 3D triangular mesh that is used in related approaches to perform the selection of clusters [Yu+12; Yu+16]. Figure 5.30 visualizes our iterative flood-filling method starting with a pre-selection of several particles in the first step.

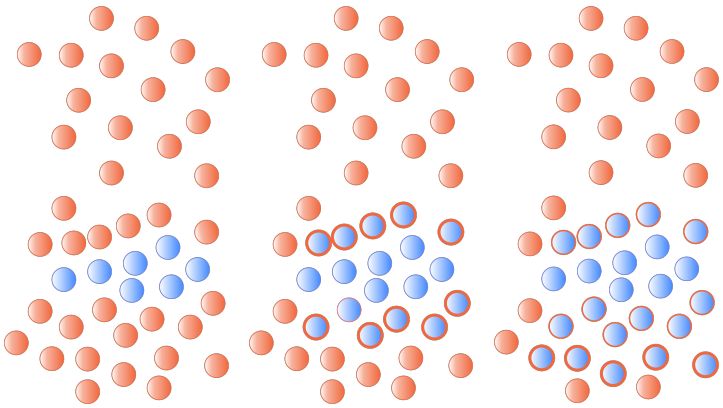

Figure 5.30: Conceptual iterative flood-filling based particle-selection process from left to right. Already marked particles are highlighted in blue. Newly marked particles (with respect to the previous iteration) are additionally highlighted with a red stroke.

During particle selection, we differentiate between *direct* and *indirect* target particles. *Direct target particles* are selected in the beginning of this phase: All particles with depth values that lie in the depth interval computed using the intention buffer are considered to be selected. Furthermore, they have to satisfy the constraint that their projection into the screen space falls into the area covered by the mask image. During this step, we also compute minimum and maximum densities ($\max(\rho_d)$ and $\min(\rho_d)$) of all direct target particles: Having this information at hand allows us to determine a density delta over all direct target particles.

$$\Delta\rho_d = \max(\rho_d) - \min(\rho_d). \tag{5.20}$$

Note that $\Delta\rho_d \geq 0$ since $\max(\rho_d) \geq 0$ and $0 \leq \min(\rho_d) \leq \max(\rho_d)$. Afterwards, we mark all *indirect target particles* iteratively by using the density delta $\Delta\rho_d$ determined using the direct target particles. Here, we iterate over all neighbors and check if the local density of this particle in comparison to its neighbors is less than $\Delta\rho_d$. The set of all indirect target particles to be marked in the next step around a given particle i is then given by

$$\text{indirect}(i) = \{j \in N(i, h_i) \mid 0 \leq \rho_j - \min(\rho_d) \leq \Delta\rho_d\}, \tag{5.21}$$

where $N(i, h_i)$ is the neighborhood function that returns all neighbors in the given local smoothing radius h_i. We stop the marking process once a fixed point has been reached and no further particles have been marked in comparison to the previous iteration. Note that ρ_d does not change during iterative marking, since it represents the density threshold given by the direct target particles which have been determined in the beginning. After marking, all selected particles based on the initial selection lasso have been detected. These can be displayed on the screen or output for further analysis, for example.

5.2.3 Complexity & Implementation Details

Runtime complexity is the limiting factor for approaches from related work. As mentioned earlier, they must explicitly instantiate certain intermediate data structures in memory. These include 3D mesh data and uniform 3D grids for computing intermediate features (e.g., for use with the Marching Cubes algorithm). Unlike these methods, which can easily take up to several seconds [Yu+12], our method takes up to tens of milliseconds, even for large datasets. This is mainly because we differentiate between operations in the screen space and operations running on the dataset itself. Moreover, our algorithms have been developed with GPUs and parallel execution models in mind. For example, any `foreach` statement can be efficiently implemented with GPU kernels in the context of the previously presented algorithms. However, it is also related to the fact that the runtime complexity of our method is only [KK18]

$$O(n \cdot k) = C_{\text{total}}, \tag{5.22}$$

where n is the total number of particles and k the number of average neighbors per particle [KK18]. In addition, our method consumes only $O(n)$ memory. To be precise, the overall complexity of our method consists of the complexity of the density estimation, the selection process, and the mask operations in screen space, and is given by [KK18]

$$C_{\text{density}} + C_{\text{selection}} + C_{\text{screen\_space}}. \tag{5.23}$$

Note that this definition includes additional steps that operate in screen space $O\left(C_{\text{screen\_space}}\right)$. These parts have a theoretical processing complexity of $O(x \cdot y)$ (see also Algorithm 6 and Algorithm 7), where x and y represent the screen resolution. Since $x \cdot y$ is not related to the dataset containing n particles, $x \cdot y$ does not dominate the runtime complexity while reasoning about all particles, as these factors are only resolution-dependent and are not directly affected by the dataset itself. Consequently, $x \cdot y$ can be seen as constant with respect to a varying number of input particles n. On the implementation side, the screen-space parts can be efficiently executed in parallel on a GPU, making this complexity term negligible in the implementation and in practice, unlike the processing steps on the dataset.

The density estimation (see Algorithm 8) and the iterative flood-filling are the most expensive operations in our processing pipeline since they operate directly on all/a subset of the particles. Reconsider the density estimation first. It loops multiple times over all particles in the dataset to find a good approximation for each h_i. In this scope, it iterates over all neighboring particles to iteratively adjust each smoothing radius [KK18]:

$$O(n \cdot k + 2 \cdot n \cdot k \cdot i_{\max}) = O(n \cdot k \cdot i_{\max}) = O(n \cdot k). \tag{5.24}$$

where i is the number iterations and k the number of neighboring particles. i is limited by $i_{\max}$ and a user-defined constant (usually $i_{\max} \in [1, 3]$). More-

over, k (the maximum number of neighbors) is closely related to N_h: We first start with a rough estimate of h_i for each particle. This may result in k being larger or smaller than N_h. After several iterations, however, we reach approximately the point where $k \approx N_h$, resulting in $k << n$. Since N_h depends on the weighting kernel, which in turn can depend on the structure of the underlying input data, it does *not* depend on the number of particles n. From a practical point of view, k is a constant factor that is included in the runtime complexity formula for the sake of completeness since its depends on the application scenario.

Similar to density estimation, flood filling involves multiple iterations over adjacent particles. In each step, we mark all neighbors that fall within our previously determined density interval (see Equation (5.21)). We continue our marking process until we reach a fixed point: There are no particles that have been newly marked compared to the previous step. So the worst case occurs when we have to mark the entire dataset, which is completely unrealistic from a practical point of view. Consider the case where we start with only a single direct target particle. We then mark all matching particles in the neighborhood of this initially marked one (a subset of its k neighbors that fall within the density interval). Consider further this involves only a single particle, and all subsequent steps also mark only a single particle. In the worst case, this leads to a runtime complexity $\in O(n)$, since we iterate over all particles. Alternatively, we could mark all $k > 1$ neighbors of all newly marked particles in each step, resulting in a total number of k^I particles marked after the Ith iteration. Since we cannot mark more particles than there are in the dataset itself, we obtain $k^I \leq n$, which yields $I \leq \log_k(n)$, where $k > 1$. From this, it follows that we are still in $O(n \cdot k)$.

Regarding memory consumption, we have to store several values (e.g., the smoothing radii h_i, the density information ρ_i and the selection state) per particle. This yields a memory complexity of [KK18]

$$O(l \cdot n) = O(n) = C_M, \tag{5.25}$$

where l is the number of data elements that we have to store besides the underlying information per particle (e.g., the position in 3D space). Note that this analysis does not take additional memory consumed for screen-space buffers and specific realizations of particle neighborhood processing methods into account . This is because our algorithm is not tied to a particular neighborhood-processing logic that may inherently be based on uniform grids or trees. Compared to other approaches from related work [Yu+12; Yu+16], this is a major advantage since they are always rely on uniform 3D grids. In other words, the neighborhood processing logic can be replaced by any suitable algorithm that provides us with an implementation for $N(i, h_i)$ (likes the ones by Groß et al. [GKK19; GKK20], see below).

In terms of implementation details, we used C# to implement our particle selection prototype. Furthermore, we leveraged the ILGPU [Kös23] compiler for all GPU-accelerated parts of the application and used Direct3D for rendering, which also integrates seamlessly with our GPU processing kernels. As mentioned above, particularly interesting is the efficient realization of the iteration logic over all neighboring particles. For our prototype we did not store neighbor lists per particle, but subdivide the simulation into a virtual uniform grid. We followed the approach by Green and Hoetzlein [Gre10; Hoe14] which assign a virtual grid index to each particle. Afterwards, we sorted all particles according to their virtual grid index. This considerably improves performance when iterating over all neighbors of a certain particle. Unfortunately, at the time of publishing this work, we had to store several additional 32-bit integers per grid cell, which is still $\in C_m$ since the grid resolution is much smaller than the number of particles. However, the use of these neighborhood models weakened our advantages in terms of memory consumption compared to related methods in general. To overcome these limitations, we later invented two highly efficient neighborhood processing algorithms [GKK19; GKK20] that significantly reduced memory consumption. We were able to reduce the overhead to a few kilobytes, regardless of the number of particles n in the dataset. This again highlights the advantages of the SSPS approach in general over related work.

5.2.4 Selection Quality and Precision Evaluation

Our evaluation consists of two main parts, namely runtime performance and selection quality. Selection quality is particularly important to compare our approach to the quality of other methods from related work [Yu+16]. Although it is not possible to compare them directly on the same datasets because they were not available, we could compare the accuracy of our evaluation with the accuracy of their methods on their datasets. We further decided to use selection scenarios that can appear in the context of real-world datasets to include an evaluation of common and challenging scenarios (in contrast to related work). However, we followed Yu et al. [Yu+16] to use artificial datasets in order to focus on specific edge cases possible to occur including scenarios used by [Yu+16]. Figure 5.31 shows our twelve evaluation scenarios that have been rendered with the help our desktop application, built for development, benchmarking and evaluation proposes (see Figure 5.32).

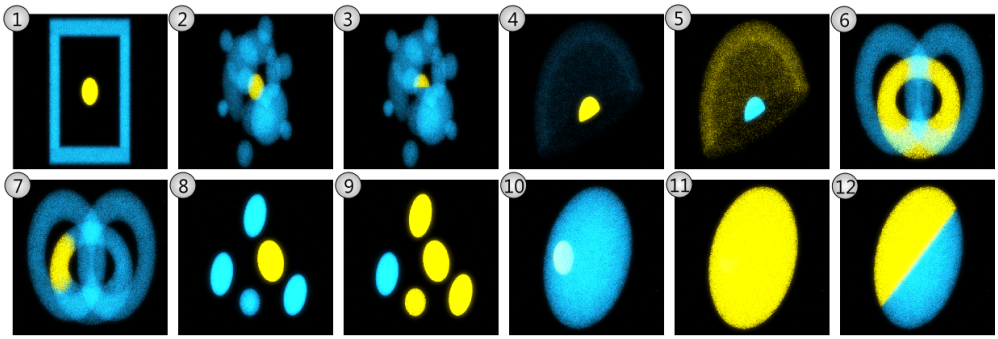

Figure 5.31: All evaluation scenarios shown to the uses using the same camera perspective. For training purposes for the users, we used the first image only [KK18].

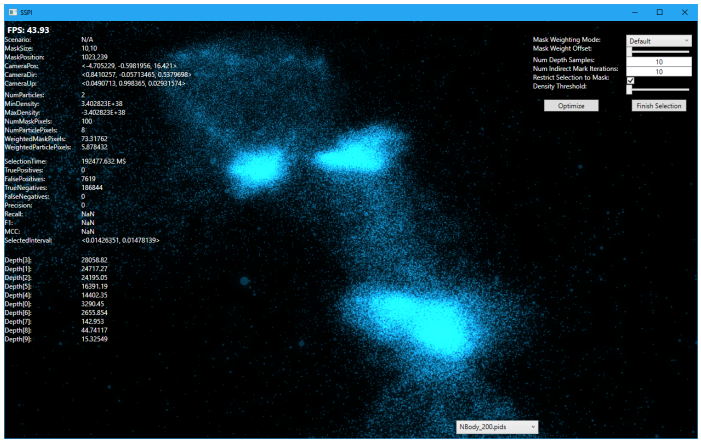

Figure 5.32: Screenshot of our desktop application used for benchmarking and evaluation of the captured selection lassos.

We used a *FullHD* screen resolution (1920×1080 pixels), in combination with a number of 16 bins for the intention buffer (see Section 5.2.1). Regarding the evaluation scenarios themselves, we used at least $\approx 150,000$ particles for the smallest and $\approx 450,000$ particles for the largest scenario in order to have a reasonable workload for our method (see Figure 5.33).

Scenario	# Particles	# Target	# Misc	# Noise
1	457K	52K	400K	5K
2	286K	10K	271K	5K
3	286K	5K	276K	5K
4	154K	105K	44K	5K
5	154K	45K	104K	5K
6	252K	82K	165K	5K
7	252K	17K	230K	5K
8	450K	105K	340K	5K
9	450K	340K	105K	5K
10	372K	105K	262K	5K
11	372K	236K	131K	5K
12	372K	315K	52K	5K

Figure 5.33: Evaluation scenario configurations for the scenarios shown in Figure 5.31 [KK18]. *Target* refers to particles that should be successfully selected using a given selection mask. *Misc* indicates particles that should not be selected and *Noise* refers to noise particles.

We also added *noise* to the dataset in the form of additional particles that affect selection accuracy and assigned each scenario to one or more of the following categories, where each category has some specific key properties that aim for specific cases as well as individual selection properties [KK18]:

- **Whole clusters**, *applies to scenarios 1, 2, 4, 5, 6, 8, 9, 10 and 11*

  This category covers the most common scenario [Yu+12; Yu+16], in which the aim is to select one or more whole particle clusters. Practically, this task is straight-forward to achieve from a user's perspective since a user usually requires a single selection lasso that spans over the desired target region. Our algorithm has to be able to map dramatically different selection lassos to the proper particle densities and thus the actually select particles.

  Focus: General mapping of selection lassos to visible areas.

- **Partial selection**, *applies to scenarios 3, 7 and 12*

  Partial selection is more difficult to realize than whole cluster selection. Our approach must pay more attention to the shape of the lasso. For example, sharp boundaries separating desired and undesired particles.

Unlike other structure-aware methods (such as that of Yu et al. [Yu+16]), we can accomplish these tasks implicitly with our generic SSPS processing pipeline without further modification. This is because we do not reason about clusters and operate on particles instead.

Focus: Precise mapping of selection lassos to visible areas.

- **Occluded selection**, *applies to scenarios 2, 3, 4, 7 and 10*

  Occluded selections are the most difficult selection use cases to achieve, as users intend to select a particle subset in the dataset that is completely or partially hidden by others. It is particular challenging for our approach since we extract information from the depth buffer that does not contain information about occluded particles.

  Focus: General mapping of selection lassos to partially occluded areas.

Precision

The precision evaluation is based on selection lassos we gathered with the help of an online user study. We preferred this online survey over an on-site study, to get gather 737 lassos in a short period of time using an interactive web application. In the scope of this survey, we showed users successively rendered scenario images (see Figure 5.31) in a counterbalanced way (except the first scenario). Before we started the actual data collection, we presented all participants several informational slides to expose them to the field of particle selection. As mentioned above, they were able to gain initial knowledge based on the first scenario. Internally, we sent the collected lassos asynchronously via the websocket protocol to our server hosted in the cloud. All actual processing tasks were performed in the cloud and the rendered image was transmitted back to the users' browsers and displayed to them in real time. The information sent back to user also included additional information about the number of particles selected, their accuracy, and average density. Note that we accepted selections as valid if a user selected at least 1000 particles. If not, we discarded the selection, displayed a message to users, and they had to try again until they made a valid selection.

Using this evaluation method, we were able to obtain selection lassos from 72 users, from which we discarded data from 5 users because they did not finish the study (reasons unknown). Consequently, we obtained data for the analysis from 67 users, of whom participants were between 20 and 46 years old ($M = 26.4, \sigma = 5.8$), while 58 were male and 9 were female [KK18]. Note that we also asked for color blindness and their input device, which revealed that we did not have any colorblind participants and all of them used the mouse as their input device. Furthermore, all of them where very experienced with their input method ($M = 4.5$ out of 5). After completing the online study, we used all raw selection masks in combination with our evaluation program (see Figure 5.32).

In analogy to Yu et al. [Yu+12; Yu+16], we used the MCC (Matthews correlation coefficient) and F1 scores to determine the quality of our selections. Both methods rely on the computation of true positives (TP, target particles that were selected), false positives (FP, default particles that were selected), true negatives (TN, default particles that were not selected) and false negatives (FN, not selected target particles). We evaluated precision by applying each lasso to its associated dataset and measuring the average $F1$ and MCC scores for each scenario (see Figure 5.34). Overall, we were able to achieve high values for both the $F1$ and MCC measurements in all evaluation scenarios. This indicates excellent precision as all scores are above 91% (after normalization). The worst results where obtained using scenarios 3 and 7, which targeted partial selections. These scenarios suffered from the issue of having similar densities in the surrounding neighborhood of the direct target particles. Achieving better results in these cases would have required further fine tuning of our default selection parameters (like the number of bins).

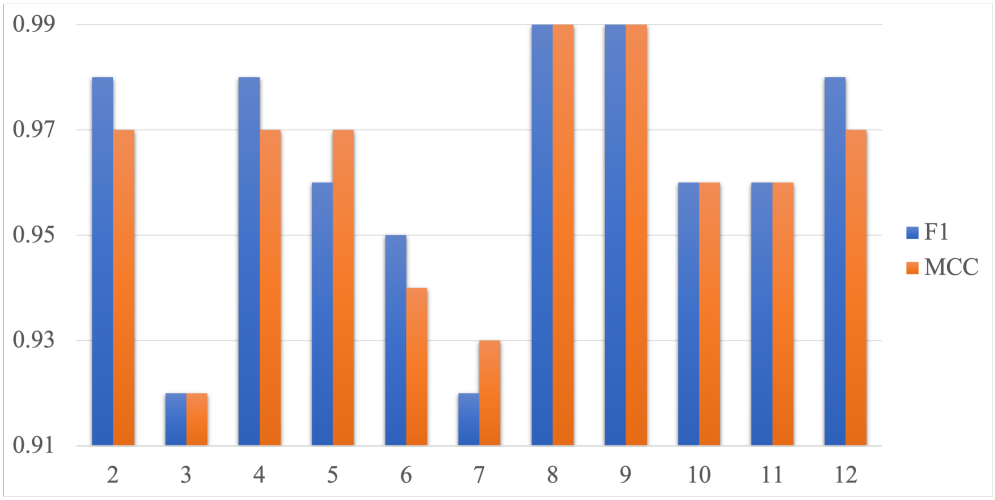

Figure 5.34: Average $F1$ (blue) and MCC scores (orange) for each scenario (higher is better, numbers published in [KK18]). Note that all numbers are normalized to the appropriate value intervals.

5.2.5 Performance Evaluation

The performance evaluation of our newly proposed method does not include a comparison to the methods presented by Yu et al. This is due to the fact that their system took several seconds in total [Yu+12] (including density estimation) with high deviations regarding the overall selection times [Yu+12]. In contrast to their approach, our method completed all tasks in the scope of milliseconds and was orders of magnitude faster in all cases (see below). Since our scenarios are chosen in analogy to their scenarios, it was still possible to compare all runtime measurements directly without having the actual numbers from Yu et al. Note that Yu et al. assumed that the density estimation step must be performed only once per dataset to practically reduce the required selection time to a few hundred milliseconds [Yu+12]. However, we made no such assumption to ensure that our method would be applicable to changing datasets (e.g., created by simulations on-the-fly). Even if we were to compare our selection performance to that using precomputed density data, we would still be at least an order of magnitude faster, while dramatically reducing memory consumption.

Our performance evaluation was run on two GPUs from NVIDIA featuring different compute capabilities [NVI23a]: GeForce GTX 980 Ti and GeForce GTX 1080 Ti [KK18][3]. To compensate for runtime variations and JIT compiler overhead, we considered one performance measure the mean execution time of 100 selections with the same selection mask randomly chosen from the collected masks of the user study [KK18]. We used a cubic spline kernel for density computations by setting $N_h \approx 42$ (a common choice for cubic spline kernels) for all tests [DA12]. The number of density approximation iterations was set to $[0, \ldots, 3]$. However, we needed up to two iterations for all evaluation scenarios to achieve a reasonable h_i approximation (see Figure 5.35). Note that the first scenario was excluded since it was used for training purposed during the user study only.

Scenario	2,3	4,5	6,7	8,9	10,11,12
# Density Iterations	1	1	2	2	2
# Average Neighbors	42	43	44	47	44

Figure 5.35: Number of density iterations required to approximate h_i for each particle and the average number of neighbors for each scenario [KK18].

[3]Note that both GPUs were state-of-the-art at the time of the publication. Therefore, we retained these measurements to present the original results. However, it is of interest to reevaluate our method on more recent GPUs in future work (see Chapter 14).

The number of density iterations depends on the number of particles and their distribution in the input dataset. This usually means that more particles require more approximation iterations. However, this rule of thumb did not hold true in all cases. Consider scenarios 2 and 6, where the latter one contains even less particles than the first. In these scenarios, the particle distributions were completely different, as there were more densely packed particles in scenario 6, which required a finer-grained adjustment of the smoothing length h_i. Related to that, our hypothesis on how to determine an initial smoothing length to start the approximation process (see Equation (5.18)) seemed to work well on all evaluation scenarios. After only two steps, we had reached a configuration that was really close to the intended optimal number of neighbors (see Figure 5.35).

Figures 5.36 and 5.37 show the runtime in milliseconds for both evaluation GPUs. The diagrams also show the density and selection proportions of the runtime via color coding. In contrast to speedup measurements, pure runtime measurements gave us the ability to to demonstrate our real-time capabilities for interactive user applications (see also Figure 5.38).

Figure 5.36: Runtime in milliseconds on the GTX 980 Ti (smaller is better, based on performance numbers published in [KK18]). Blue: Density approximation. Orange: Selection process.

In most cases, the runtime was dominated by our density approximation phase (highlighted in blue in our diagrams). However, in some case, the selection phase was more time consuming. An example for this is scenario 4 in which the density estimation performed a single step only but the selection phase required more iterations to complete our flood-filling based approach. In scenarios 11 and 12, the runtime was about evenly distributed between the selection and density phases, which is related to the large number of target particles that needed to be selected during flood filling.

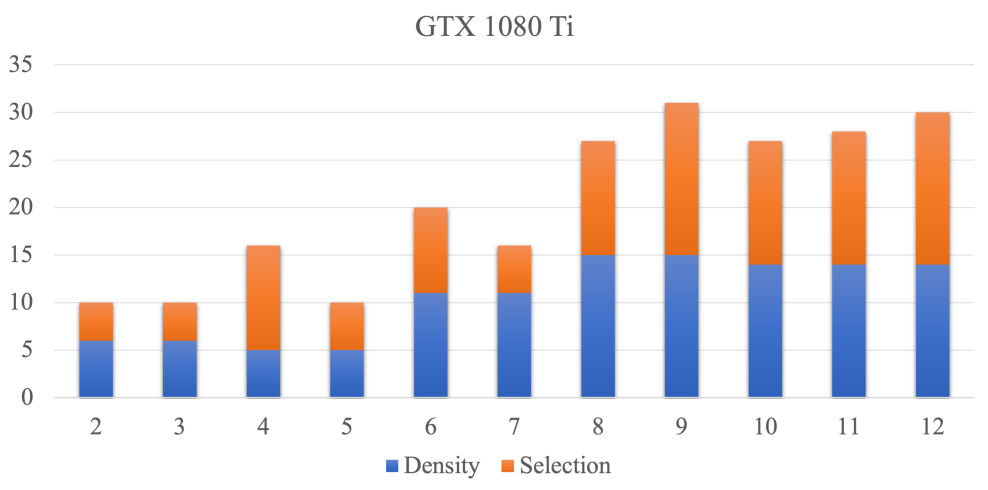

Figure 5.37: Runtime in milliseconds on the GTX 1080 Ti (smaller is better, based on performance numbers published in [KK18]). Blue: Density approximation. Orange: Selection process.

Comparing both GPUs against each other reveals that the GTX 1080 Ti was considerably faster in all cases (up to 2.0×) compared to the GTX 980 Ti. This was due to the fact that the more recent GTX 1080 Ti had improved memory bandwidth and processing capabilities. Furthermore, this also shows that our algorithm benefited from more recent GPU advancements by leveraging the available hardware capabilities in a very efficient way.

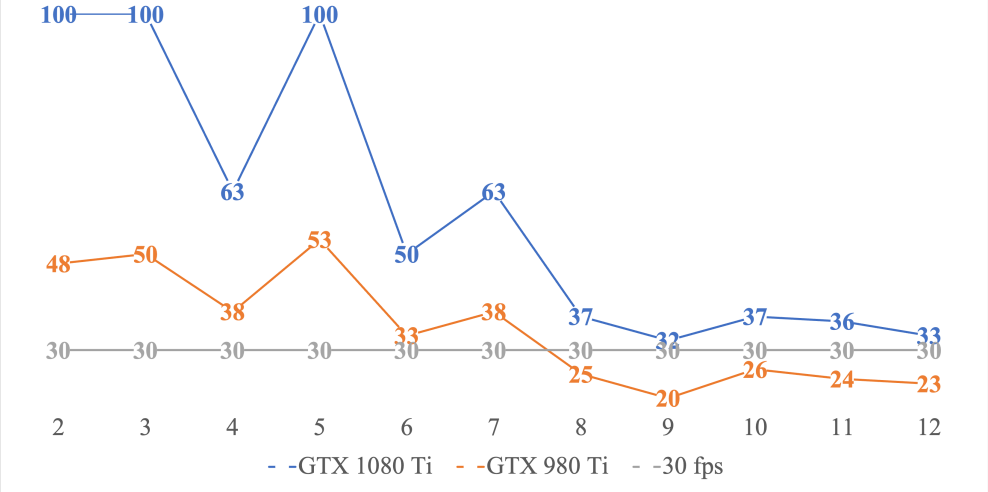

Figure 5.38: Runtime in *FPS* in comparison to a real-time application running at 30 *FPS* (higher is better, based on performance numbers published in [KK18]).

Looking closer to differences between scenarios reveals that doubling the number of particles (e.g., scenario 4 compared to scenario 10) results in doubling the runtime (approximately). This is a result of the fact that more particles in these cases required more density and selection iterations. However, doubling the memory bandwidth and/or the computational performance on a GPU results in halving the runtime in the best case.

Figure 5.38 shows all runtime measurements for both GPUs while adding a 30 frames-per-second (*FPS*) line to the diagram, which can be considered real-time for interactive applications accepting user input [KK18]. As shown, the more recent GTX 1080 Ti was able to perform more than 30 full selection processes on the datasets without any pre computation. In comparison to these results, the older GTX 980 Ti was considerably slower and could not perform 30 selections per second on all scenarios. Note that this was only a theoretical benchmark to show the effectiveness of our approach, since performing multiple selections per second has not been an intended use case. However, the ability to complete the selection as quickly as possible provides a high level of convenience in the selection process for the end user. In sum, all measurements confirm the real-time capabilities of our method and its well scalability to real-world datasets by handling thousands of particles in the scope of milliseconds.

CHAPTER 6

IMPROVING PERFORMANCE OF GENERIC MASSIVELY-PARALLEL SIMULATIONS

After presenting innovations from the domain-specific field of particle-based simulation, this chapter focuses on the extension and generalization of generic simulations. In contrast to the set of domain-specific problems, we cannot benefit from optimization potential on the algorithmic-mathematical side, since we do not have access to deep domain knowledge about the underlying problems. In the following sections, we consider simulations given by a set of components C_i (see Figure 6.1) that are executed sequentially. Each component C_i represents a small logical module that must be applied to complete a full simulation step (denoted by the backedge in Figure 6.1).

Figure 6.1: Generic simulation flow of a single simulation step with multiple components C_0 to C_4 [KGK20a]. The purple backedge indicates that C_4 is the last component in a single step and that the next step starts again with the execution of C_0.

We introduce high-level simulation concepts in the first section. Section 6.2 presents approaches to improve occupancy and memory throughput for compute- and memory-bound generic simulations. These concepts can be further combined and improved using adaptive time-stepping methods for arbitrary simulations presented in Section 6.3.

6.1 Simulation Basics

Reconsider the APBF algorithm for simulating iteration-adaptive position-based fluids (see Algorithm 5 of Section 5.1.3). In this case, each loop iterating over all particles represents a logical processing part (see Figure 6.2). A single simulation step contains two nested loops, one of which performs multiple pre-stabilization iterations and the other iteratively solves the density constraint (blue backedges). After updating all positions p and velocities v, a full step is completed and we begin the next step by applying all external forces (purple backedge).

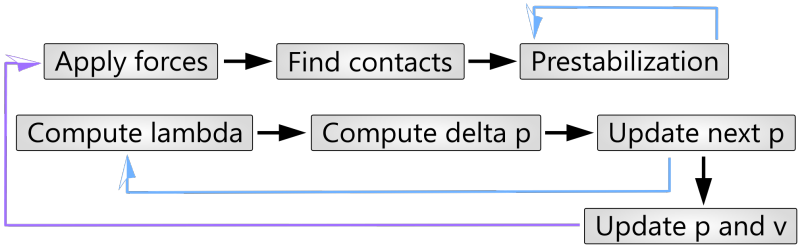

Figure 6.2: High-level simulation workflow of the APBF algorithm (see Algorithm 5) including two nested loops (blue backedges).

As shown in this example, a simulation step may contain nested loops. These loops can either be considered as standalone logical components or fully unrolled to form a loop-free execution flow within a single simulation step. To simplify all subsequent visualizations, formulas, and algorithms, we will consider a simulation step to be loop-free in the remainder of this chapter. When unrolling loops or merging them into a single component is not an option, all of the approaches presented can be applied recursively from the innermost loop to the outermost loop.

Implementing such a high-level component in a parallel way on the GPU can be achieved via a grid-stride loop (see Algorithm 9) to avoid unnecessary dynamic warp/group dispatching overhead. Each component is assumed to loop over a sequence of elements (like particles) to which we can apply the component in parallel. The number of elements to iterate over is referred to as *value range*. We spawn as many thread groups as possible to achieve maximum occupancy of the device [NVI]. Each thread computes its globally unique element index j, initializes the ith component C_i with the element index of the current element j. Then, each thread applies its uniquely configured component instance using the current time step size, denoted Δt without loss of generality. Note that $range(C_i)$ returns a padded number of elements to iterate over. The value returned by $range(x)$ is padded to a multiple of the group size to avoid thread divergence in the body of the grid-stride loop. This is generally not a strict requirement, but it allows safe use of group-wide synchronization primitives.

Algorithm 9: Grid-stride loop algorithm for a component C_i [KGK20a]

```
  /* Perform a grid-stride loop over the padded value range
     of C_i                                                    */
1 for j := global index to range(C_i) step by (grid size * group size) do
  |  /* Initialize component by loading elementwise
  |     information and apply component                       */
2 |  c := C_i::Init(j);
3 |  c.Apply(Δt);
4 end
```

In practice, each component can be implemented as a *class* (in object oriented programming) that follows a specific interface. We used an abstract pseudo-code interface definition in Listing 6.1 to present general features that a component must provide. Here, the type *TSourceBuffer* is a structure containing required data pointers to all source buffers in global GPU memory. This enables components to load all necessary information from the global memory during the initialization phase. Similarly, the *TTargetBuffer* type contains all pointers to each target buffer in global memory. In the case of double buffering (see Chapter 3), data pointers in these structures point to different memory buffers to avoid read-write data dependencies between component executions. Note that this interface is also generic in terms of the datatype modeling domain-specific values of Δt.

Listing 6.1: A pseudo C#/C++ version of an interface for a component [KGK20a].

```
1  // A generic component interface that depends on the actual type to
       implement a domain-specific time delta per step.
2  // TSourceBuffer and TTargetBuffer refer to user-defined structures that
       hold references to read-only/write-enabled sections of GPU memory.
3  // TComponentImplementation will be instantiated with a specific type that
        implements this interface.
4  interface IComponent<
5    TTime,
6    TSourceBuffer,
7    TTargetBuffer,
8    TComponentImplementation>
9    where TTime : struct
10   where TSourceBuffer : struct
11   where TTargetBuffer : struct
12   where TComponentImplementation :
13     struct, IComponent<
14       TTime, TSourceBuffer, TTargetBuffer, TComponentImplementation>
15 {
16   // Initializes the internal fields of this component by loading
         information from global memory.
17   static TComponentImplementation Init(
18     in TSourceBuffer source,
19     int elementIndex);
20
21   // Applies this component by executing the intended instructions.
22   void Apply(TTargetBuffer target, TTime deltaT) const;
23 }
```

As an example, we consider a *distance constraint* from the field of position-based dynamics [Mül+07; Mül08]. The purpose of this constraint type is to ensure a certain distance between two particles in 3D space. For simplicity, we omit the mathematical definition and the implementation of the position delta correction to solve a distance constraint instance. Listing 6.2 defines the required miscellaneous data structures for sources and targets. As mentioned previously, the source structure contains pointers to immutable regions in global memory. This ensures that all density constraint instances can be initialized in parallel. The target structure contains a single pointer to a region in global memory that contains all position delta corrections for each particle. It also defines the *DistanceConstraintData* structure, which contains all the information needed to instantiate a constraint.

Listing 6.2: Data structures to implement a distance constraint from a position-based dynamics framework [Mül+07; Mül08] in pseudo C#/C++ code.

```
1  // Read-only source buffers.
2  struct SourceBuffers
3  {
4    // All particles in the simulation.
5    public const Particle* Particles;
6
7    // All distance constraints.
8    public const DistanceConstraintData* DistanceConstraints;
9  }
10
11 // Write-enabled target buffers.
12 struct TargetBuffers
13 {
14   // All position deltas that will be accumulated during a solver
         iteration.
15   public Vector3* PositionDeltas;
16 }
17
18 // Represents necessary information to instantiate a single distance
       constraint.
19 struct DistanceConstraintData
20 {
21   // The index of the left and right particles affected by this distance
         constraint.
22   public int LeftParticleIndex;
23   public int RightParticleIndex;
24
25   // The intended distance between the two particles in 3D space.
26   public float Distance;
27 }
```

Listing 6.3 shows an example implementation of the *DistanceConstraint* structure using the data structures from Listing 6.2. It uses seconds for time stepping and loads particle data from global memory into the highly efficient register space as part of the *Init* method. The component application first calculates the position corrections *pdLeft* and *pdRight* for the left and right particle, respectively. It then uses atomic operations to accumulate position-delta corrections. Atomic operations help us to ensure the integrity of all corrections

stored in global memory, since theoretically multiple density constraints can affect the same particle. At the same time, the parent component-application algorithm (see Algorithm 9) executes multiple component instances in parallel.

Listing 6.3: A sample component corresponding to the component interface from Listing 6.1 in pseudo C#/C++ code using the data structures from Listing 6.2. It implements parts of a distance constraint from a position-based dynamics framework [Mül+07; Mül08]

```
struct DistanceConstraint : IComponent<
  float,              // Operates on seconds
  SourceBuffers,      // Custom source data structure
  TargetBuffers,      // Custom target data structure
  DistanceConstraint> // Our constraint structure
{
  // The underlying constraint data.
  private DistanceConstraintData data;

  // Data of the left and right particles affected by this distance
      constraint.
  private Particle left;
  private Particle right;

  // Load required particle data from memory.
  static DistanceConstraint Init(SourceBuffers source, int elementIndex)
  {
    // Load the specific constraint-data instance from global memory.
    var data = source.DistanceConstraints[elementIndex];

    // Create new constraint instance.
    var constraint = new DistanceConstraint();
    constraint.data = data;

    // Load particle data for this constraint instance from global memory.
    constraint.left = source.Particles[data.LeftParticleIndex];
    constraint.right = source.Particles[data.RightParticleIndex];
    return constraint;
  }

  // Applies density-constraint projection while using the given deltaT in
       seconds.
  void Apply(TargetBuffers target, float deltaT) const
  {
    // Compute position corrections using data from both particles (left
        and right).
    var pdLeft = ...;
    var pdRight = ...;

    // Accumulate results for both particles using atomic operations to
        avoid race conditions between parallel constraint evaluations.
    Atomic.Add(
      ref target.PositionDeltas[data.LeftParticleIndex],
      pdLeft);
    Atomic.Add(
      ref target.PositionDeltas[data.RightParticleIndex],
      pdRight);
  }
}
```

Adaptive Simulations

Similar to particle-based simulations, the concept of adaptive generic simulations exists in generalized forms without reasoning about domain-specific knowledge. Figure 6.3 shows a simple way to realize time-step adaptive component-based simulations relying on our description published in [KGK20a]. It is based on the idea of using an initial step-size estimation phase. Each component is logically divided into two parts: One that focuses on the actual component logic, and one that computes the step size for the upcoming step (referred to as S_i and C_i).

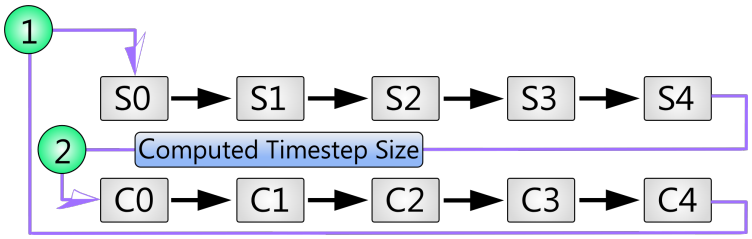

Figure 6.3: Generic simulation flow of a single simulation step using adaptive time step sizes [KGK20a]. In addition to a set of logical components (C_0 to C_4 in this case, see Figure 6.1), this approach uses a set of step-size components S_0 to S_4. These components will be executed first to determine a step size for the actual simulation components.

In practice, this separation is realized by two different methods defined in the same interface without the need to write logically differentiated components. Listing 6.4 presents an abstract component definition that supports the separation of concerns of S_i and C_i shown in Figure 6.3. There are generally two conceptual ways to realize this approach from an implementation point of view. The first possibility is to maintain a fixed *deltaT* for all components and determine a number of steps (a multiple of *deltaT*) to perform. This version involves a function *ComputeNumSteps*, which is invoked on an initialized component instance and returns the number of steps. Furthermore, the *Apply* method is adjusted to accept a determined number of steps as a parameter (*numSteps*). The second possibility is to adapt *deltaT* using a method similar to *ComputeDeltaT*. However, this version does not require an adaptation of the *Apply* method. Note that we need to determine the *minimum step size over all components* [KGK20a] to get a safe lower bound for the next step size in all cases (see Figure 6.4, given in the diagram over the first phase using the S_i components).

This places an additional requirement on the *ComputeDeltaT* method, since the actual type of *deltaT* (*TTime*) must supply a commutative and associative min function. This is not a limitation or requirement in the first place, where we work with 32-bit or 64-bit integer values that support commutative and associative minimum operations.

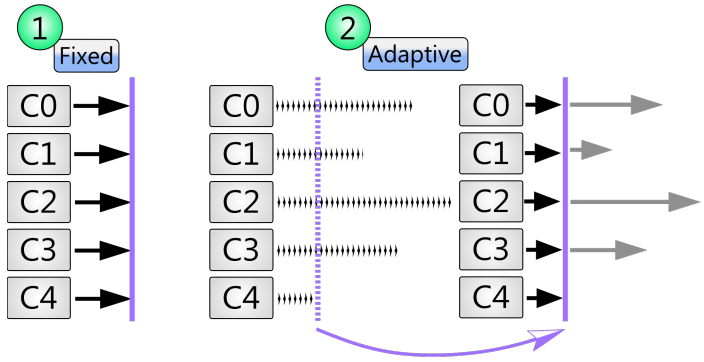

Figure 6.4: A set of components ($C_0 \dots C_4$) and their step sizes visualized via black arrows [KGK20a]. If all components have the same step size, all arrows will have the same length (left, 1). In an adaptive scenario, we have to determine a *compatible step size* (indicated via the purple line) for all components (right, 2). This leads to the use of the minimum step size of all components in the next step. It further avoids constraint violations that can destabilize the simulation and lead to wrong results. Here, the intended step sizes for all components are visualized using gray arrows.

Listing 6.4: A time-adaptive component interface in pseudo C#/C++ code based on Listing 6.1 [KGK20a].

```
1  // Defines a generic time-dependent component.
2  interface IAdaptiveComponent<...>
3    where ...
4  {
5    // Initializes the internal fields of this component by loading
         information from global memory.
6    static TComponentImplementation Init(...)
7
8    // ------------------------------------------------------------------
9    // Possibility 1: Compute a number of steps.
10   // ------------------------------------------------------------------
11
12   // Computes the number of steps this component can perform in the next
         simulation step.
13   int ComputeNumSteps(TTime deltaT) const;
14   // Applies this component by executing the intended instructions.
15   void Apply(TTargetBuffer target, TTime deltaT, int numSteps) const;
16
17   // ------------------------------------------------------------------
18   // Possibility 2: Compute the step size.
19   // ------------------------------------------------------------------
20
21   // Computes the step size this component can perform in the next
         simulation step.
22   TTime ComputeDeltaT() const;
23   // Applies this component by executing the intended instructions.
24   void Apply(TTargetBuffer target, TTime deltaT) const;
25 }
```

Algorithm 10: Our simulation loop using adaptive time steps (based on work published in [KGK20a])

```
   Input: #steps, #components, inputBuffer, outputBuffer
1  maxStepBuffer := allocate int on GPU;
2  for s := 0 to #steps step by stepSize do
       /* Copy required time-step information to the GPU    */
3      CopyToGPU(#steps - s, maxStepBuffer);
       /* Compute common adaptive step-size for all
          components                                         */
4      for i := 0 to #components do
5          Compute minimum step size for C_i using S_i;
6      end
7      CopyFromGPU(maxStepBuffer, out stepSize);
       /* Apply all components                             */
8      for i := 0 to #components do
9          Apply C_i using stepSize;
10     end
       /* Optional:  swap buffers for double-buffering
          applications                                       */
11     Swap inputBuffer, outputBuffer;
12 end
```

Algorithm 10 shows an implemented simulation loop designed for adaptive time steps. This version explains a high-level simulation loop using the first approach based on adapting the number of steps with the help of a fixed *deltaT*. A memory buffer on the GPU is allocated to transfer data regarding the upcoming time step size from the GPU to the CPU parts of the application and vice versa. First, the transfer buffer must be initialized with the remaining number of steps (the maximum number of steps to reach). Then, all step size calculation kernels S_i are executed to calculate the global minimum step count. This is achieved by instantiating each component individually, invoking the *ComputeNumSteps* method from Listing 6.4, and performing a global min reduction within the same kernel. Next, we need to synchronize these calculations while fetching the contents of the exchange buffer back into CPU memory to get the number of steps. Finally, we can apply all components C_i with the computed step size and proceed with the next simulation step. Optionally, we swap input and output buffers in the case of double buffering (see Section 3.1). Note that this algorithm can be automatically specialized by a compiler to avoid unnecessary nested loops iterating over all components, further improving performance.

6.2 Parallel Simulations of Multiple States using Interpreters

As outlined earlier, we consider generic simulations based on components that implement a particular interface. Going a step further than the interfaces from the basic introduction in Section 6.1, it turns out that a common use case is to skip certain elements instead of applying each component to every element. From a developer's perspective, this conditional check can be implemented directly in the *Apply* function of the presented interfaces (see Listing 6.1 and Listing 6.4). However, if we make the apply check an explicit method, we can directly distinguish between pieces of code that affect the condition and other pieces that operate on the application logic. This technique allows programmers to express additional domain knowledge that is also beneficial when writing code transformations and compilers that work with the component interfaces described. A straightforward extension of the component interface is shown in Listing 6.5.

Listing 6.5: Predicated component interface in pseudo C#/C++ code based on Listing 6.1.

```
1 // Defines a generic predicated component.
2 interface IPredicatedComponent<...>
3   where ...
4 {
5   // ...
6
7   // Checks whether this component instance can be successfully applied.
8   bool CanApply() const;
9
10  // ...
11 }
```

Using this domain knowledge allows us to extend Algorithm 9 with an *if condition* to skip elements to which the component cannot be applied (see Algorithm 11). As outlined in this simple algorithm, the actual condition check is done as part of the surrounding algorithm and not hidden in the component implementation itself.

Algorithm 11: Component algorithm for C_i using *if-guards* [KGK20b]

```
  /* Perform a grid-stride loop over value range of C_i     */
1 for j := group index to range(C_i) step by (grid size * group size) do
  |  /* Initialize component by loading element info        */
2 |  c := C_i::Init(j);
  |  /* Check component precondition for current element    */
3 |  if c.CanApply(j) then
  |  |  /* Apply component                                  */
4 |  |  c.Apply(Δt);
5 |  end
6 end
```

Furthermore, the condition within the loop causes divergent control flow with respect to other threads running in the same warp and group. Note that divergence can in turn cause noticeable performance degradation due to non-optimal occupancy of the target device (see Chapter 2, [KGK20b]). This behavior is also illustrated in Figure 6.5, which shows the previously presented if-guarded execution algorithm in the scope of an imaginary warp with eight lanes. In this sample, four threads pass the initial *CanApply* test (block `A`) and enter the application part of each component (denoted by block `B` in the diagram). However, the pattern presented realizes a separation of concerns where a component does not have to deal with the issue of how to efficiently skip a single element with respect to other component instances (running in different threads, see Chapter 2).

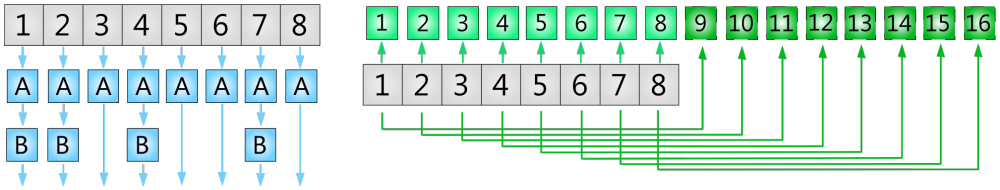

Figure 6.5: Divergent control flow in the scope of a conceptual warp with eight lanes using the simple if-guarded execution algorithm (left) [KGK20b]. Intended coalesced memory access pattern with respect to all threads (right).

In fact, the structure of the described components is related to formal operational semantics of a program, which are given in the form of inference rules. In this context, an inference rule R_i is given by [KGK20b]

$$R_i = \frac{\Gamma \vdash \mathrm{Pre}_1 \ldots \Gamma \vdash \mathrm{Pre}_n}{\Gamma \vdash \mathrm{Con}}, \tag{6.1}$$

where Pre_j, $j \in [1, \ldots, n]$ refers to all preconditions that must be satisfied in the current context Γ [KGK20b]. Furthermore, Con denotes the consequence that holds after the rule application of R_i [KGK20b]. Successively updating the context Γ by applying all rules represents a program execution using the underlying operational semantics represented by all rules. In addition, we get an updated context Γ' containing all updates caused by changes/updates of the rule. When applying all rules until we reach a fix point (there are no rules that can be applied any more, since their preconditions are not fulfilled), this yield [KGK20b]:

$$\{R_1, \ldots, R_n\}_\Gamma \Rightarrow \{R_1, \ldots, R_n\}_{\Gamma^1} \Rightarrow \ldots \Rightarrow * = \{R_1, \ldots, R_n\}_{\Gamma^k}. \tag{6.2}$$

After reaching a fix point, the finally determined context Γ^k contains by definition all updates by all rules.

This theoretical basis can actually be formed with the help of *if-conditions*. In analogy to Equation (6.1), multiple pre-conditions are mapped to conjunctions of imperative checks. A context Γ in this scope is given by the current

state of the program including all of its *variable-value* bindings. In addition, we consider rules that are tested against different object instance to live within a single program state. We denote the number of instances to which a rule R_i applies as the *value range* of R_i. This range is an interval of integer values in reality ($[0, \ldots,$ value range $- 1]$ [KGK20b]), which will be mapped to specific object instances in memory.

A simple parallel implementation on a GPU would apply all rules sequentially using separate kernels. Consider a sample implementation of an imaginary rule R_1 that is executed on an accelerator (see Algorithm 12). It assumes the presence of multiple states, in which each thread group processes a single state on its own (line 1). Afterwards, we leverage the pattern of a *group-stride loop* [NVI14] (lines 2–6) to parallelize over R_1's value range. Finally, all preconditions must be checked prior to evaluating the rule (line 4) in the loop body.

Algorithm 12: Simple parallel rule-execution algorithm using multiple states [KGK20b]

```
   Input: state information S
   /* Load state information for current state s          */
1  s := S[grid index];
   /* Execute a blocked loop with respect to R1's value
      range                                               */
2  for j := group index to range(R1) step by group size do
   |  /* Test preconditions                               */
3  |  if Condition(s, i) then
   |  |  /* Evaluate on current state                     */
4  |  |  Evaluate(s, i);
5  |  end
6  end
```

In practice, Algorithm 12 can be used as a template for specialization purposes in order to improve performance. A further advantage of this approach is that is implicitly achieves device-wide synchronization. Consequently, this automatically circumvents race conditions that may arise when executing different rules in parallel.

An alternative to multiple kernels are large combined kernels consisting of multiple rules to be executed (see Algorithm 13). In analogy to the previously presented algorithm, it uses group-stride loops to iterate over all values in individual value ranges of all rules involved (lines 2–6 and lines 8–12). Important to mention are the group-wide synchronization barriers after executing each rule (e.g., R_1 in lines 2–6). These barriers cause all state-changing memory operations to be committed, ensuring that all threads within the thread group have access to all changes. As in the case of Algorithm 12, Algorithm 13 can also be used in combination with specialization approaches to improve performance in practice.

Although both high-level approaches seem to be generally applicable, they suffer from occupancy deficiencies on GPUs. Consider a scenario in which a thread group spans over multiple warps (as explained in Chapter 2). In this case, thread divergence can occur even at the coarse-grained group level: If some threads have already completed processing a rule-specific loop, they would have to wait until the other threads in the group reached the same barrier (see Chapter 2). Unfortunately, this behavior is implicitly introduced by rule-specific precondition checks that protect the actual execution logic of each rule.

Algorithm 13: Simple parallel rule-execution algorithm for multiple states & rules [KGK20b]

```
   Input: state information S
   /* Load state information for current state s          */
1  s := S[grid index];
   /* Execute a blocked loop with respect to R1's value
      range                                               */
2  for j := group index to range(R1) step by group size do
   |  /* Test R1's preconditions                          */
3  |  if R1.Condition(s, j) then
4  |  |  R1.Evaluate(s, j);
5  |  end
6  end
   /* Wait for all changes to be committed               */
7  group barrier;
   /* Execute a blocked loop with respect to R2's value
      range                                               */
8  for i := group index to range(R2) step by group size do
   |  /* Test R2's preconditions                          */
9  |  if R2.Condition(s, j) then
10 |  |  R2.Evaluate(s, j);
11 |  end
12 end
   /* Evaluate additional rules...                       */
```

6.2.1 Leveraging Thread Compaction and Coalesced Memory Accesses in the Presence of Multiple States

Our approach to increase the overall utilization, and thus, the actual performance of an interpreter-based parallel simulation is the use of *thread compaction*. This concept is a well-known method that has been successfully applied to many domains [FA11; RE13]. The high-level idea is shown in Figure 6.6. Reconsider the simple example of a rule precondition check in block A that might cause some of the threads to skip the execution block B. Due to the nature of the lock-step execution scheme, some of the threads will be idling and have to wait for all other threads to complete block B (see also Chapter 2). Using thread compaction allows us to redistribute the workload of different block B instances among all threads. The goal is to gain free (unused) warps that can be exchanged by the warp dispatcher engine to swap in previously stalled warps waiting in the queue. Note that this concept is also similar to *stream compaction* [BOA09], in which contiguous streams of elements are compressed by filling gaps in the stream (see Chapter 4). The main difference with thread compaction is that it is not sufficient to shuffle individual data items between members of a thread group. Instead, we have to move all required state and value bindings to the appropriate lanes in each warp.

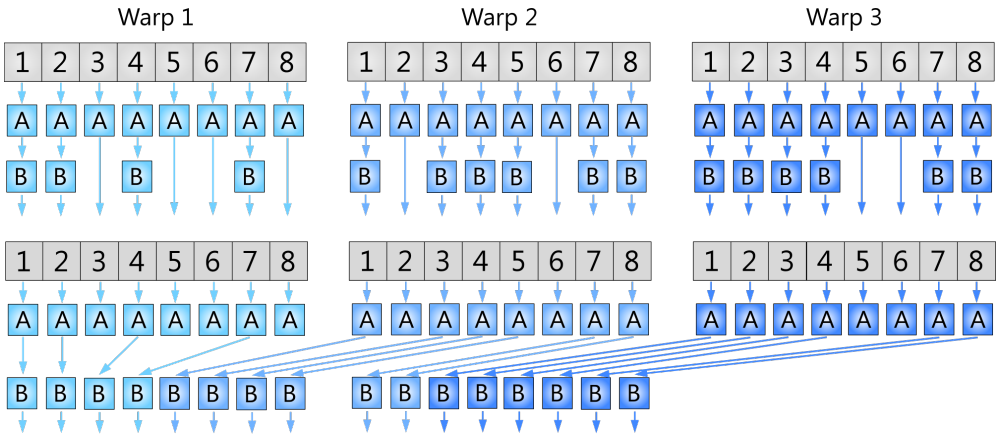

Figure 6.6: Comparison between the absence (top) and presence (bottom) of thread compaction in the context of a thread group consisting of three warp threads [KGK20b]. All threads execute a program involving thread divergence causing some threads to skip the execution of block B. Using thread compaction (bottom) allows us to release a single warp leaving more scheduling flexibility to the dynamic warp dispatcher. This helps to improve overall occupancy.

Based on research from related work, it is fairly straight forward to come up with an implementation of thread compaction operating on a single logical processing state. For instance, in the case of ray-tracing, a single frame is usually rendered at once while all scene data is shared with all threads [Wal11].

Most of the rules we have seen in real-world applications are based on a unified control flow within the actual execution code. Hence, thread compaction on the application-algorithm level is sufficient to overcome most conceptual deficiencies in terms of utilization in our cases.

More important in this scope is to come up with a well-chosen thread-group configuration, which has a huge influence on the application of thread-compaction in practice. Since this method implemented in software needs threads to share information between each other, this approach is limited to threads in a single thread group. Moreover, each of the group-stride loops iterating over the individual value ranges (of each rule) mainly depends on the number of threads per group, as well as the different value ranges. Consider the case where there are many threads in each group and the individual ranges are quite small in comparison to the number of available threads. In this case, there will be a low workload per thread leading to a suboptimal resource occupancy in many cases. However, the consideration of multiple states makes this task even simpler. Consider a *large-scale problem* involving thousands of parallel states, which can usually be found in the scope of massively-parallel optimization problems [KGK19a; KGK19c; KGK20b]. The characteristics of such problems in general is the fact that a single state also consists of hundreds to thousands of individual data elements (e.g., object instances and variables) that must be tracked and updated. Applying the concept of a rule-based interpreter to these multi-state domains results in large value ranges over which the group-stride loops in our algorithms have to iterate. Important to note is that related work determined that value ranges are usually relatively small (< 4096) in comparison to the number of states and the total number of elements per state even in very sophisticated scenarios [KGK20b]. This is due to the fact that these domains involve many rules which are be applied to different parts of each state.

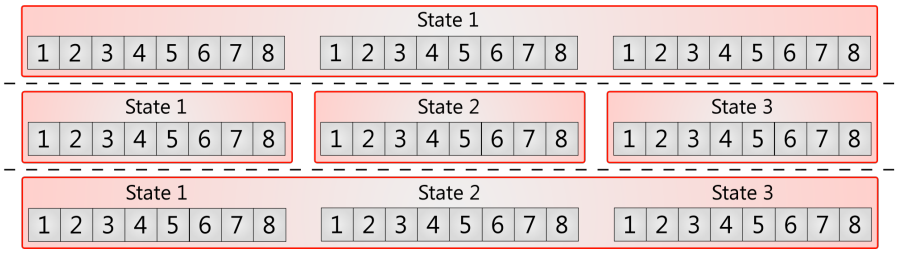

Figure 6.7: Multiple possibilities to realize processing of multiple states in the context of different thread group setups (red regions) [KGK20b]. Top: A straight-forward method to use a single thread group per state to process including many warps. Middle: A special case of the version at the top by using a single warp per thread group. Bottom: Our hybrid approach that combines large thread groups built from multiple warps, each processing a different state.

Figure 6.7 visualizes multiple thread-group configurations [KGK20b]. As discussed before, large thread groups will often cause a waste of resources. The same holds true for small thread groups (e.g., group sizes equal to the warp size of the accelerator), which causes additional work for the warp dispatcher and may also exceeded the number of parallel groups per multiprocessor on many GPU architectures [NVI23a; AMD19]. We propose a different setup which uses large thread groups processing multiple states in the scope of each warp. In contrast to the previously mentioned approaches, we subdivide each thread group imperatively into multiple "chunks" (sub groups) ourselves. This reduces the amount of work for the GPU/driver to handle all parallel groups, enhances our abilities to apply thread compaction (even across state boundaries), and improves overall efficiency.

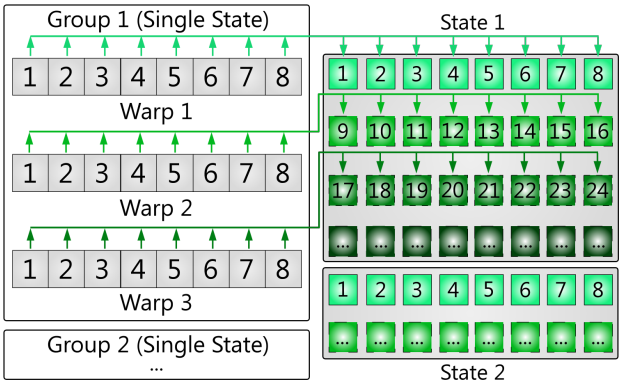

Figure 6.8: Access pattern of Algorithm 12 and Algorithm 13 to different elements in global memory living in different states [KGK20b]. Note that warps in each group can issue coalesced accesses to successively stored elements in each state.

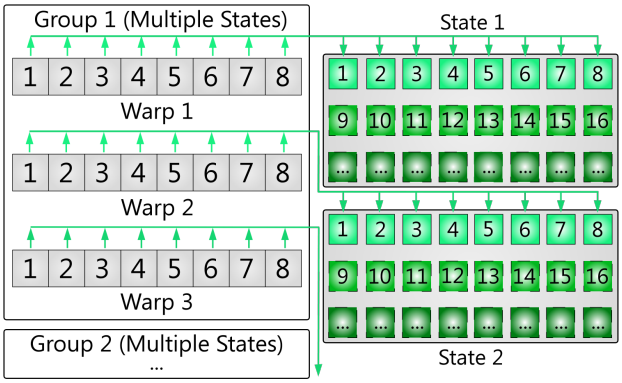

Figure 6.9: Access pattern for processing multiple states within the context of a single thread group using the memory layout shown in Figure 6.8 [KGK20b]. Since each warp operates on a different state, loads and stores will not be coalesced on the thread-group level.

The real challenge to further improve performance is to use our approach to perform efficient load and store operations in global memory (see Section 3.1). Consider Figure 6.8, which visualizes the access pattern of using a single large thread group per state spanning multiple warps. In this case, each thread group has contiguous memory access to the individual elements in memory to a single state. It causes accesses from different thread groups to be non-coalesced with respect to neighboring states. Although these effects can be hidden by spawning many thread groups, this issue becomes more serious when reducing the group size in order to avoid thread-utilization issues.

Figure 6.9 also illustrates the memory layout used in Figure 6.8. Here, we again store per-state information after each other in memory in the presence of our intended processing scheme. As shown in the diagram, our approach to process multiple states within large thread groups is not suitable for coherent memory accesses using this layout. Although the access pattern works well on the warp level, it is not possible to ensure coherent accesses for each thread on the group level. Introduced latency can generally be hidden as well, but group synchronization barriers prevent us from hiding all memory IO costs.

Our approach to achieve a well-suited memory layout is shown in Figure 6.10. Instead of storing all elements per state after sequentially in global memory, we apply a specific striding/blocking scheme that is based on the current warp size. This ensures that each warp within each group can access all state-dependent element information using coherent memory accesses most of the time. Note that the overall data stride between the first element and the last element of the first state is equal to the warp size times the number of total states.

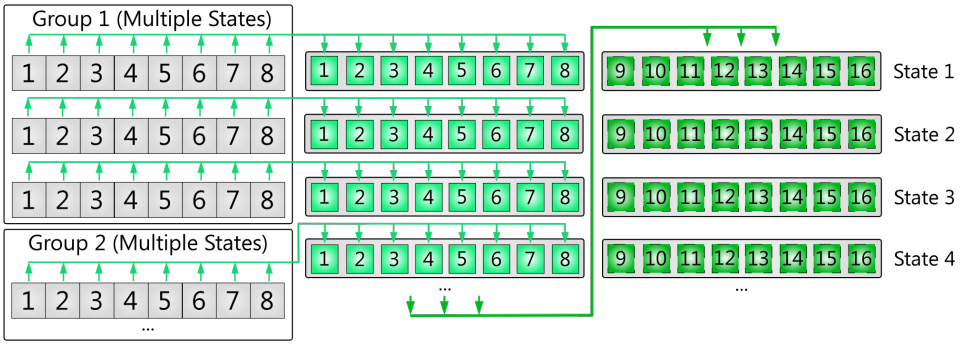

Figure 6.10: Our access pattern using a blocking/striding scheme for state-dependent element information [KGK20b]. Data is rearranged in a way that all elements accessed by a single warp are stored next to each other in global memory. Therefore, all the first "warp-size many" elements of each state are adjacent to each other, followed by the next "warp-size many" elements from all other states. This guarantees coalesced memory accesses for all threads in all groups. However, our layout requires more expensive address computations to realize group-wide and state-aware striding.

6.2.2 Algorithms

A pseudo-code version of our GPU-kernel method is presented in Algorithm 14. It assumes a single warp per state in the context of multiple warps per thread group (as presented in Figure 6.7). Moreover, it leverages the principle of prefix sums to implement thread compaction across all threads in a single group (refer to Line 19 for more details).

Algorithm 14: Our Execution Algorithm [KGK20b]

```
   Input: state information, #states per group
1  shared := shared memory int[group size];
2  sourceStateIndex := group index * group size /
3      #states per group + thread index / warp size;
   /* Apply rule R_1                                          */
4  for nextIndex := (thread index % warp size) to value range of R_1 step by
     warp size do
5      currentIndex := nextIndex;
6      stateIndex := sourceStateIndex;
       /* Check rule condition                                */
7      enabled = R_1.Condition(stateIndex, currentIndex);
       /* Compute prefix sum of all enabled threads           */
8      (offset, threadOffset) := prefix sum (enabled, shared);
       /* Perform thread compaction in shared memory          */
9      if enabled then
10         shared[offset−1] := (stateIndex, currentIndex);
11     end
12     group barrier;
       /* Check whether we are an active thread               */
13     if thread index < threadOffset then
           /* Get state and value indices from compacted
              shared memory                                   */
14         (stateIndex, currentIndex) := shared[thread index];
           /* Evaluate rule with the determined state and
              value indices                                   */
15         R_1.Evaluate(stateIndex, currentIndex);
16     end
17     group barrier;
18 end
   /* Evaluate additional rules ...                           */
19 ...
```

Initially, we allocate two 16-bit integers per thread to store value and state indices in shared memory. This results in a single 32-bit allocation per thread (line 1). Note that this allocation is properly chosen to avoid bank conflicts on modern GPUs [NVI23a]. Next, we compute the logical source-state index the current thread is associated with (lines 2–3). This is required since we handle multiple logical states per thread group.

The evaluation of all rules is designed to work in a sequential manner. Without loss of generality, we start by processing the first rule $R1$. In this scope, we iterate over all values in the value range of $R1$ by performing a blocked loop using a stride equal to the warp size of the current GPU. Since querying the value range yields the same value in all states, and thus, for all threads in the current group, the loop does not introduce thread divergence. This is particularly important as we use group-wide barriers and warp shuffles in the loop body. Otherwise, this can result in unintended side effects like kernel freezes.

At this point of the algorithm, all threads are considered to be active and evaluate the condition of the first rule (line 7). This results in either a value of 1 (*true*) or 0 (*false*) for each thread (stored in the variable *enabled*). We call threads that evaluated the condition test to *true*, *enabled* threads. To actually compact all enabled threads, we use the well-known principle of prefix sums [NVI14], or more precisely, inclusive prefix sums. The prefix-sum algorithm is applied on the group-level while using our shared-memory array from the beginning, yielding two values: *offset* and *threadOffset*. The *offset* variable points to the next location in shared memory to which the current state and value-range indices should be written. The variable *threadOffset* contains the total amount of all threads in the group that are considered *enabled*. Note that this number can be easily retrieved from the last thread of the group as a byproduct of the prefix-sum algorithm.

To realize the actual compaction step, we have to store our current state and value-range indices to the element at *offset*-1 in shared memory (since we use inclusive prefix sums, lines 9–11). This is done in each *enabled* threads, as we want to skip disabled threads that did not pass the condition test. Afterwards, we have to wait for all threads to commit their updates to shared memory using a group-wide barrier (line 12). The final (and main) step of this execution algorithm is to change the context information of the first *threadOffset* number of threads. By checking whether the current group-wide thread index is less than *threadOffset*, we separate *enabled* threads from the others (lines 13–16). The only operations left are to get the (new) current state and value-range indices from shared memory and applying rule $R1$ using this information. The barrier immediately after this if block is also very important, as it prevents overrides by other threads that skip the if block and already enter the next loop iteration. Further rules will be evaluated using the same principle one after another.

Implementation Details

This section covers implementation details of our method shown in Section 6.2.1 for evaluation and re-implementation purposes. In addition, Algorithm 15 provides a read-to-use pseudo-code snippet that implements an inclusive prefix-sum computation designed for Algorithm 14.

Regarding the evaluation, we inlined the prefix-sum implementations into the main algorithm and used C++ to develop our prototype. All GPGPU computing tasks had been realized using CUDA and compiled with the CUDA 10 SDK for evaluation purposes (see Section 6.2.3). Due to the constraints of GPU kernels being limited to shared memory in order to share information as efficiently as possible among members of a single group, this imposed several restrictions in practice. On the latest GPU architectures available at the time of release, the number of states per group were constrained to 32 states on NVIDIA GPUs, as the warp size was 32 and there could be up to 1024 threads per group [NVI23a] (see also Chapter 2).

Algorithm 15: Implementation details of the prefix-sum computation [NVI14; KGK20b]

```
   Input: value, shared
   Output: offset, threadOffset
1  offset := warp prefix sum (value);
   /* Only the last lane of each warp writes to shared
      memory                                                  */
2  if lane index + 1 = warp size then
3  |  shared[lane index] = offset;
4  end
5  group barrier;
   /* Use first warp to complete the prefix-sum computation
      */
6  if lane index = 0 then
7  |  shared[lane index] := warp prefix sum (shared[lane index]);
8  end
9  group barrier;
   /* Adjust offsets using updated shared-memory contents  */
10 if lane index > 0 then
11 |  offset := offset + shared[lane index - 1];
12 end
13 threadOffset := shared[warp size - 1];
14 group barrier;
```

As outlined in the previous paragraph, Algorithm 15 contains information on how to realize a suitable inclusive prefix-sum implementation for our algorithm. Here, this algorithm accepts a current thread-local value (used to compute the prefix-sum information) and a reference to a section in (shared) memory to store intermediate results. This pseudo-code algorithm is based on related

work [NVI14; KGK20b] and uses warp-shuffle operations to efficiently solve the group-wide prefix-sum task on the warp level. Initially, each warp computes a warp-local prefix sum resulting in thread-local offsets (line 1). Information about the last (right most) lane in each warp is then propagated to the first warp in the group via shared memory (line 2–4). A group-wide barrier in line 5 ensures that all warps in the group have written their offset information to the indicated memory section.

Afterwards, the first warp in the group performs a warp-wide prefix sum again to update the warp-local offset information (line 6–8). After waiting for the first warp to complete this step (line 9), we can adjust all offsets in all warps (except the first one) (lines 10–12). The total number of *enabled* threads after performing thread-compaction *threadOffset* can be directly determined from the information shared by the last thread in this group (line 13). Note that the last step needs to be a group-wide barrier (line 14) to avoid reuses of the given memory section. If this algorithm operates on a distinct section of memory and other concurrent updates to this section can be avoided, this barrier will be redundant.

6.2.3 Performance Evaluation

We evaluated our method using different memory layouts and varying work loads while running benchmarks on two GPUs from NVIDIA featuring different compute capabilities [NVI23a]: GeForce GTX Titan X and GeForce GTX 1080 Ti [KGK20b][1]. The core part of the evaluation is based on a comparison of our approach to the commonly used and straight-forward version from Algorithm 13. In order to evaluate different work load configurations for each rule, we relied on artificially generated rules based on matrix multiplications. This decision made the results easily understandable and even more important domain independent and reproducible. Furthermore, the decision was inspired by the fact that modern heuristically-based simulation/optimization systems often rely on machine-learning models to guide the decision process [KGK19a; KGK19b; KGK19c; KGK20a; KGK20b].

Conceptually, an actual evaluation-rule implementation in the scope of our evaluation was therefore given by the matrix multiplication $A \times B$, where $A \in R^{M \times N}$ and $B \in R^{N \times O}$ [KGK20b]. Matrix multiplications were well suited in this case since they represent fundamental neural network mechanics [KGK19a]. Following Hunger [Hun07], a single rule then performs $2MNO - MO$ floating-point operations. In order to have direct control over the number of operations, we used a surrounding loop that iterates *load*-factor of iterations. This made adjustments of the matrix dimensions unnecessary which would have involved changes of memory allocations/layouts during a single run of the benchmark suite. Moreover, it allowed us to conveniently analyze the general scaling behavior when increasing/decreasing the computational load. The main part of the evaluation used a default *load* factor of 20 iterations to create some basic workload. Note that each measurement was determined computing the median of the execution time over 100 runs to minimize external influence factors.

All memory layouts used for evaluation purposes where previously presented in Section 6.2. In the scope of this section, we will refer to the various layouts as follows:

- **A** represents the layout from Figure 6.8, which uses coalesced memory accesses within each state,
- **B** represents non-coalesced memory accesses by using the transposed memory version of A (see Figure 6.9), and
- **C** represents our newly introduced memory layout for realizing coalesced memory accesses within and across state boundaries (see Figure 6.10).

In addition we defined a *divergence rate d* [KGK20b], which allowed us to control whether every dth thread will have a divergence on average. For each

[1] Note that both GPUs were state-of-the-art at the time of the publication. Therefore, we retained these measurements to present the original results. However, it is of interest to reevaluate our method on more recent GPUs in future work (see Chapter 14).

state, a value was chosen randomly by drawing a uniform sample from the integer interval $\in [0, \ldots, d-1]$. Here, each non-zero value ($d \neq 0$) caused a thread to pass the rule-specific precondition checks on the one hand. On the other hand, a value of $d = 0$ skipped the actual rule application for the current thread. In order to mimic real-world experience from practice in terms of common divergency patterns [KGK20b][2], we set d alternately to 2 and 3. To refer to different configurations, we used combinations of the memory layouts and the divergence rate used. For instance, memory layout A combined with a divergency rate $d = 2$ is referred to as *A2*.

We began the actual evaluation by comparing speedups of memory layouts A and C to B using the two divergence rates $d = 2$ and $d = 3$. Figures 6.11 and 6.12 show the speedup comparisons on both evaluation GPUs using the simple execution method from Algorithm 13. The diagrams show the speedup behavior across multiple state configurations in terms of the number of states and different value ranges per rule. This allowed us to clearly analyze effects on the scaling behavior when introducing more load by either increasing the number of states, the value ranges, or even both at the same time. We considered 2048 states to be a small problem to solve [KGK20b] and 16384 states as a slightly more complex problem to deal with. Similarly, we considered 1024 to be a fairly reasonable value range to iterate over per rule (the number of objects to handle per rule) and 4096 to be slightly more [KGK20b].

Since layout B could already be considered the least optimal one due to its non-coalesced memory accesses, the results were not surprising. The more rule applications there were to process, the better the overall speedup was compared to the B layout. The smallest state configuration (2048 states and a value range of 1024) was at least twice as fast on the GTX Titan X and at least four times faster on our GTX 1080 Ti. This even held true for different divergency rates, which did not impact the overall speedup significantly when comparing measurements of *A2* to *A3* and *C2* to *C3*.

Note that we lost a factor of 2.0× in terms of the relative speedup when going from the GTX Titan X to the GTX 1080 Ti. This was due to the fact that the more recent GTX 1080 Ti did a much better job hiding non-coalesced memory accesses introduced by memory layout B. Hence, this resulted in a considerably better compensation of choosing the "wrong" memory layout for this job. Our proposed memory layout C did not perform very well against layout A in all cases. We could see speedups as low as 0.88× on the GTX Titan X (slowdowns, for $d = 2$) in the worst case and speedups up to 1.06× in the best case. The GTX 1080 Ti performed slightly better using memory layout C resulting in speedups of 1.03× in the worst case and 1.07× in the best case. This could be seen as negligible improvements in general. It was related to the fact that the address computation for each chunk of data to read from/write to global memory was more expensive in the case of layout C than computations

[2]See also Chapter 12 for more detailed project information.

using layout A. Thus, the older GTX Titan X was affected more due to its less powerful processing units.

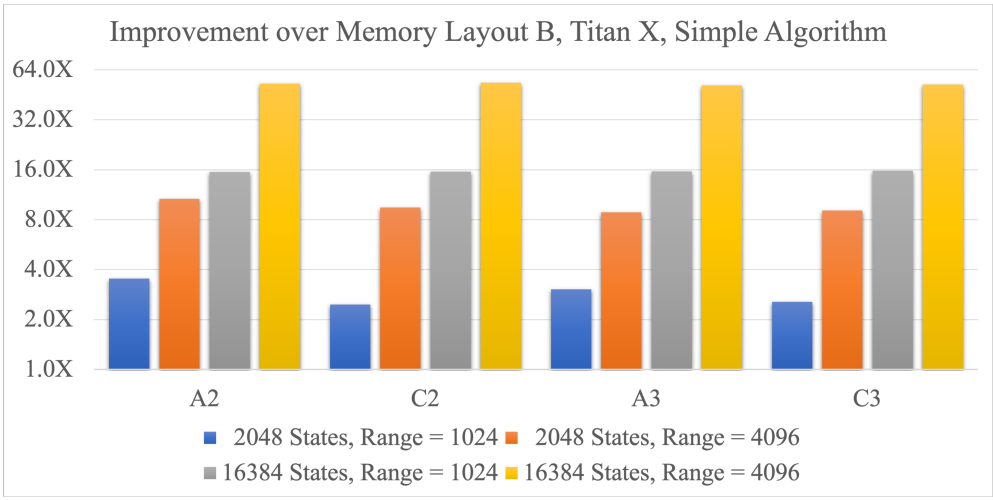

Figure 6.11: Relative speedup comparison of memory layouts A and C to B using two value ranges, two state configurations, and the simple execution algorithm on the GTX Titan X (log2 scaling, higher is better, based on performance numbers published in [KGK20b]). An actual performance comparison between the simple and our algorithm is shown in Figure 6.13.

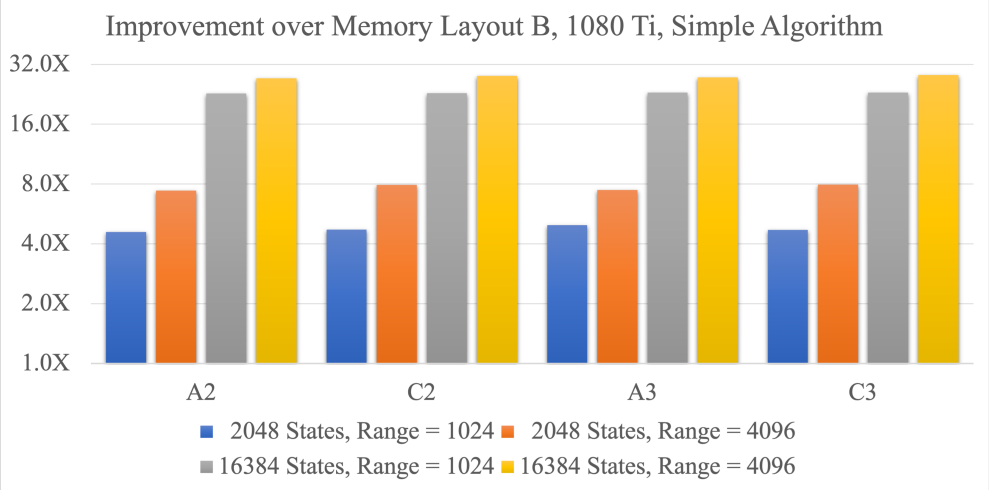

Figure 6.12: Relative speedup comparison of memory layouts A and C to B using two value ranges, two state configurations, and the simple execution algorithm on the GTX 1080 Ti (log2 scaling, higher is better, based on performance numbers published in [KGK20b]). An actual performance comparison between the simple and our algorithm is shown in Figure 6.14.

Similar to measured speedups on the older GTX Titan X using the simple approach, we also lost a relative speedup of 2.0× on the more recent GTX 1080 Ti using our method. This is again related to the structure of the problem to be solved, which does not benefit significantly from additional processing capabilities. As seen before, changing the divergency rates did not change the relative speedup compared to memory layout *B*. However, memory layout *C* did result in a slight degradation of performance on the GTX Titan X, whereas it always caused a reproducible speedup of up to 1.14× on the GTX 1080 Ti using our algorithm.

Moving on to actual performance comparisons of the simple algorithm and our approach, unveiled tremendous speedups. Figures 6.13 and 6.14 shows the speedup comparison of our approach to the simple algorithm using two value range and two state configurations. On the GTX Titan X, we measured speedups ranging from 1.9× to 3.3× (see Figure 6.13). Increasing the value ranges for all rules did not yield any further speedups when used with our method in combination with any divergency rate $d \neq 0$. This was due to the fact that the GTX Titan X already reached their maximum occupancy very quickly using our method.

Additionally, the relative speedup using our proposed memory layout *C* was considerably better ranging from 1.1× to 1.25× compared to the speedups achieved with traditional layout *A*. That was because of the functionality of our algorithm: Using thread-compaction moved threads closely together, frees computational resources, and causes increased utilization of the available processing units. In addition, we also benefited from evenly distributed address calculations that compensated the impact of more complex address calculations in these cases. However, the more states and computational load we created on the GPU, the more speedup we could see on the GTX Titan X. Increasing the number of states by 4× resulted in an additional speedup increase of $\approx 1.45\times$ in the case of $d = 2$. Increasing our divergency rate to $d = 3$ caused more rules to pass precondition checks before applying the actual rule logic still caused an additional $\approx 1.3\times$ uplift in this case.

The more recent GTX 1080 Ti also benefited from our proposed memory layout *C* due to its additional and more capable computing units (see Figure 6.14) on the one hand. On the other hand, using the *A* layout only resulted in speedups ranging from 1.03× to $\approx 1.1\times$. This was related to the fact that we maxed out the available utilization on the GPU quickly and caused the performance improvements to be capped at $\approx 3.5\times$. It also explained the minor performance improvements comparing scenarios with $d = 2$ to the ones using $d = 3$. Since the maximum load of the device was already almost reached in most cases, packing more threads together no longer resulted in any significant increase in speed. Generally speaking, increasing the overall input load by 4×, resulted in a performance improvement of 2× (comparing *A2*, 2048 states and a range of 1024 to *A2* using 2048 states and a range of 4096 values). This was true until the maximum occupancy was reached.

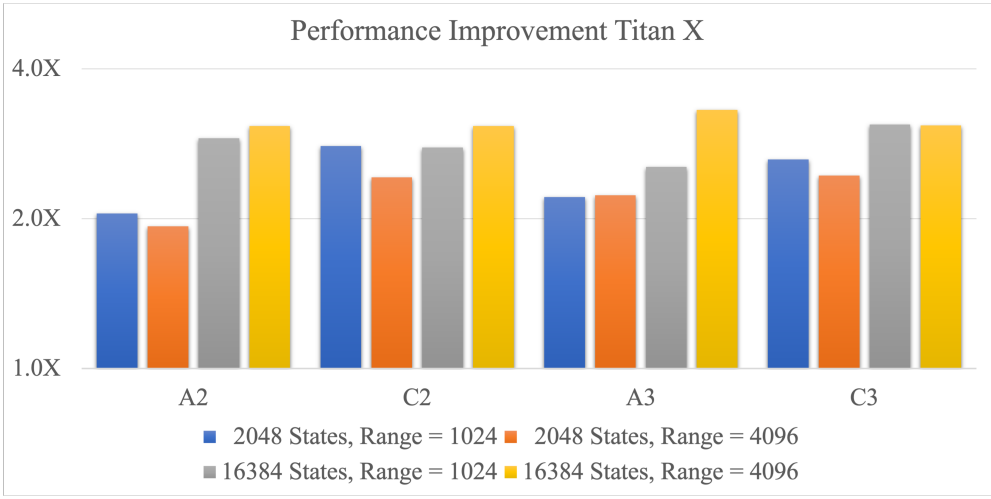

Figure 6.13: Speedup of our algorithm compared to the simple one on the GTX Titan X using two value ranges and two state configurations (log2 scaling, higher is better, based on performance numbers published in [KGK20b]).

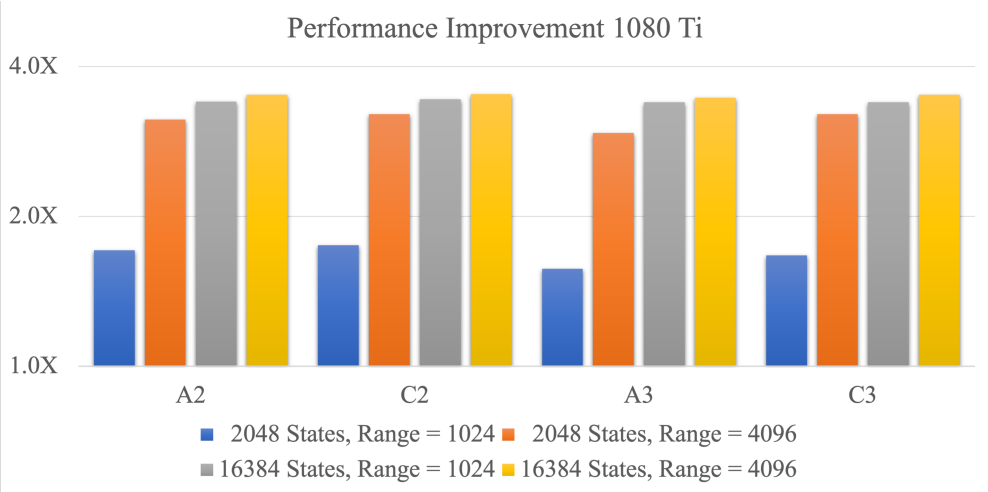

Figure 6.14: Speedup of our algorithm compared to the simple one on the GTX 1080 Ti using two value ranges and two state configurations (log2 scaling, higher is better, based on performance numbers published in [KGK20b]).

Moreover, figures 6.15 and 6.16 show the speedup comparisons of memory layouts A and C to B on both evaluation GPUs using our algorithm. In analogy to the previously shown measurements, we also evaluated the same state configurations in terms of the number of states and the value ranges being used.

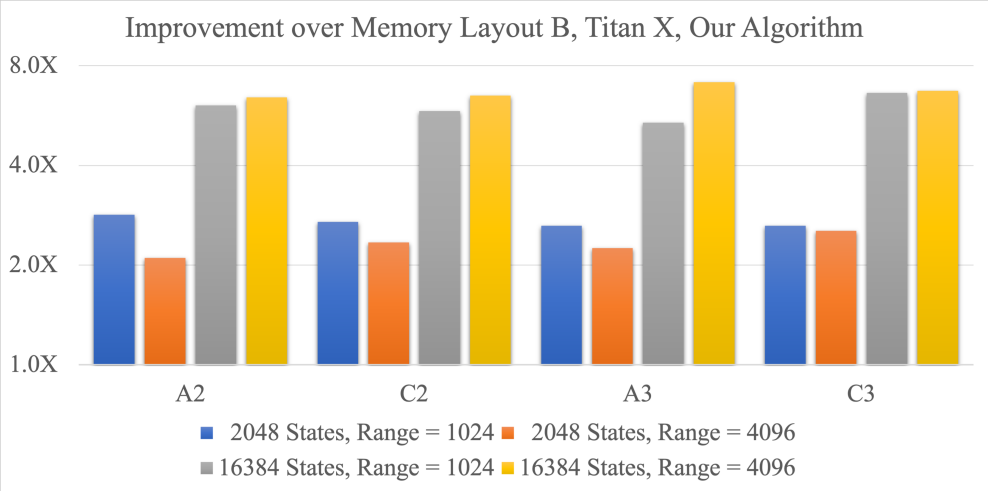

Figure 6.15: Relative speedup comparison of memory layouts A and C to B using two value ranges, two state configurations, and our execution algorithm on the GTX Titan X (log2 scaling, higher is better, based on performance numbers published in [KGK20b]).

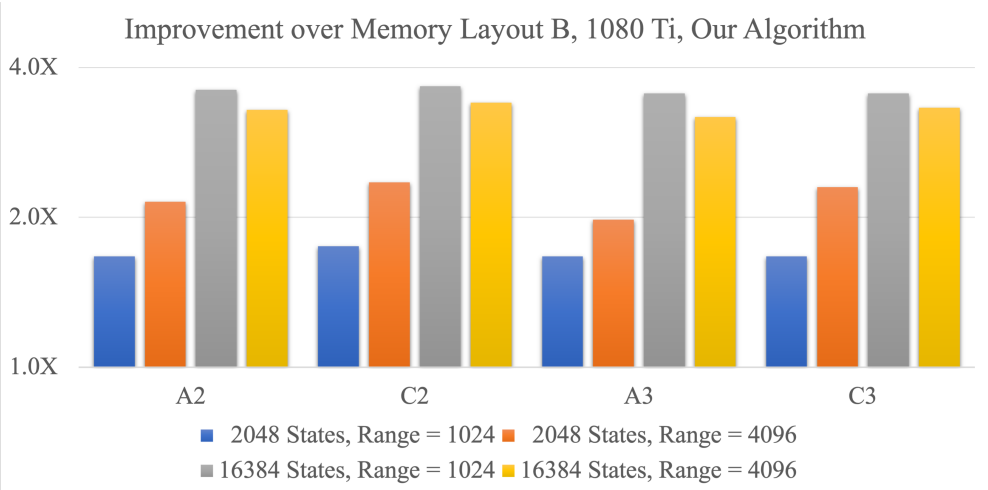

Figure 6.16: Relative speedup comparison of memory layouts A and C to B using two value ranges, two state configurations, and our execution algorithm on the GTX 1080 Ti (log2 scaling, higher is better, based on performance numbers published in [KGK20b]).

6.3 Adaptive Time Stepping for Generic Simulations

The previous sections introduced a way to improve the efficiency of generic component-based simulations using thread-compaction techniques. An alternative, the use of adaptive time steps, was also discussed in the introduction to this chapter to reduce the computational load of simulations. The main problem with using adaptive time stepping without domain expertise is that it can easily lead to slowdowns. This is because the calculation of the time steps itself requires additional effort compared to simulations with fixed time steps. Therefore, the adaptive version can only be faster overall if the time steps are larger on average and the slowdowns caused by the time-step calculation itself are compensated.

Consider a simulation with a reference step size of 1 (the smallest possible step size that fulfills all requirements). Assume further, that this imaginary simulation supports integer-based time steps only. Since we need to calculate the minimum step size of all simulation components in order to execute an actual step, we need to reason the probability of a single component returning the smallest step size. This probability is given by $P = 1 - p^n$ [KGK20a], where n is the number of components and p is the probability of a single component returning a step size > 1 [KGK20a]. In other words, the probability grows significantly when increasing the number of simulation components, and thus, can easily lead to performance penalties in general.

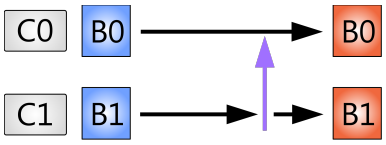

Figure 6.17: A visualization of two components C_0 and C_1 and their associated source (blue, B_0^s and B_1^s) and target buffers (orange, B_0^t and B_1^t) [KGK20a]. In this scenario, C_1 has a read-data dependency on C_0 (purple arrow). Also, C_0 can do a large time step (long black arrow), while C_1 can only do a smaller step, relying on the temporary information provided by C_0. In order to allow C_0 to perform its large step and C_1 to conceptually perform the same step, C_1 needs to access interpolated information from B_0^s and B_0^t. This allows us to perform a single step for C_0 and two smaller steps for C_1 using interpolated information at the point in time indicated via the purple arrow.

Our approach to overcome this conceptual limitation, is visualized in Figure 6.17. The idea is to conceptually unblock components performing larger time steps by providing intermediate results to components limited to smaller time steps. This is achieved via interpolated results using source and target buffers of each simulation step. Using interpolation eliminates data dependen-

cies between different components enabling larger time steps in comparison to the previously presented approach.

In order to reason about the simulation, its behavior and the interpolation function, we define the following formal specification of a simulation system. We refer to the current simulation state (containing all required information) to as S. Here, a single stance instance S is updated by a set of components C_i (as before), where $i \in [0, \ldots, |\text{Comonents}| - 1]$. Since each component operates on a subset of all information contained in a state S, we refer to the data subset touched by component C_i via S_i. The state-update equation is then given by [KGK20a]

$$S_i(T + \Delta t) := C_i\left(S_i(T), \Delta t\right), \tag{6.3}$$

where T is the current simulation time and Δt is the upcoming time step size used for this simulation step [KGK20a]. Our method includes an interpolation function I (domain specific and user provided) for approximating intermediate state subsets $S_i^I(T + \Delta u)$, where $\Delta u \in]0, \ldots, \Delta t[$ [KGK20a]:

$$S_i^I(T + \Delta u) := I_{\Delta u}\left(S_i(T), S_i(T + \Delta t)\right). \tag{6.4}$$

Extending the supported input interval of Δu to $[0, \ldots, \Delta t]$ yields [KGK20a]

$$S_i^I(T + \Delta u) := \begin{cases} S_i(T + \Delta u) & \Delta u = 0 \wedge \Delta u = \Delta t \\ I_{\Delta u}\left(S_i(T), S_i(T + \Delta t)\right) & \text{else,} \end{cases} \tag{6.5}$$

which means that all evaluations of $S_i^I(T+x)$, where $x \notin \{0, \Delta t\}$ will introduce approximation errors depending on/caused by I. Due to the use of I to estimate a future state, we get [KGK20a]

$$S_i(T + \Delta u) \approx S_i^I(T + \Delta u) \text{ and} \tag{6.6}$$

$$S_i(T + \Delta u) \approx I_{\Delta u}\left(S_i(T), S_i(T + \Delta t)\right) \tag{6.7}$$

by limiting Δu to $\Delta u > 0 \wedge \Delta u < \Delta t$. Reconsider Figure 6.17, in which one of the components needs to do two steps in time, here $\Delta l > 0$ and $\Delta o > 0$, which add up to the maximum step size Δt (given by $\Delta t = \Delta l + \Delta o$) [KGK20a]. Assuming two arbitrary components C_i and C_j, where $i \neq j$ and C_j needs to access state information from S_i, this yields [KGK20a]:

$$S_i(T + \Delta t) = C_i\left(S_i(T), \Delta t\right) \tag{6.8}$$

$$S_j(T + \Delta l) = C_j\left(S_i(T), S_j(T), \Delta l\right), \tag{6.9}$$

$$S_j(T + \Delta t) = C_j\left(S_i(T + \Delta l), S_j(T + \Delta l), \Delta o\right) \tag{6.10}$$

$$\approx C_j\left(S_i^I(T + \Delta l), S_j(T + \Delta l), \Delta o\right) \tag{6.11}$$

$$\approx C_j\left(I_{\Delta l}\left(S_i(T), S_i(T + \Delta t)\right), S_j(T + \Delta l), \Delta o\right). \tag{6.12}$$

As formally derived, we need to apply the interpolation function only once to estimate the current state of S_i at time step $T + \Delta l$ based on pre-computed information from $S_i(T + \Delta t)$ by using C_i. However, the introduced approximation function I can cause simulation deviations. Therefore, the interpolation function must be carefully chosen depending on the problem- and domain-specific properties of the simulation [Ada+07; HHK09; Ihm+10; KGK20a]. An alternative to using analytically defined simulations is the trend to replace or enrich them by using neural networks [LOL19; Lew21]. This also means that the interpolated intermediate results can be approximated using similar techniques. Last but not least, the neural-network approaches also take advantage of approximations by definition to compute the outcome of a simulation step while introducing acceptable deviations [LLK19; Lew21].

In contrast to learned simulation approximations, our method is a generic approach to add adaptive time-stepping to arbitrary simulation systems by using an algorithmic method designed for parallel architectures. Particularly important in this scope is to apply the interpolation function to the appropriate buffers. For this reason we must be able to access the source and target buffers (see also Figure 6.17) to compute interpolated results. Consequently, our method relies on double buffering (see Section 3.1) while we have to pay attention to the fact whether a component is executed an even or odd number of times. The situation described is shown in Figure 6.18: C_1 is executed two times and its initially used source buffer (blue box) will contain the actual state updates. As these updates are not written into its target buffer (orange box), we must copy the data from source to target buffer in an additional step. After copying the data into target buffer, the updates are perfectly in sync with the data of all other components. This also makes it impossible to use the source buffer after the second step for further interpolation purposes. Hence, a copy of the original contents of the source buffer is required to ensure valid interpolation results.

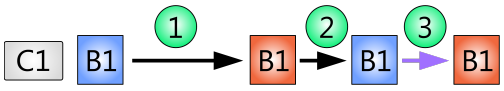

Figure 6.18: Two simulation steps performed by C_1 based on the scenario of Figure 6.17 to C_0's step size [KGK20a]. To begin with, C_1 commits its changes into its target buffer (1, orange box). Next (2), C_1 is reapplied as it reads from the designated target buffer and writes its changes to the initial source buffer (blue box). Without any further operation, other components either need to be aware of a different source buffer in upcoming steps or the source information will be invalid. To overcome this issue, we decided to integrate a final commitment phase which copies data into the intended target buffer in such cases (3).

The information needed to automatically apply the interpolation functions to the right places is determined via concepts inspired by static program analysis [Pie02; NNH10]. The basic idea is to reason about the data-dependency graph defined by all component memory accesses (see Figure 6.19). Since the dependency graph naturally contains at least a single cycle, it is usually the task of a domain expert to reason about the primary cyclic dependency in order to decide on a *first component* in a schedule. This makes the differentiation between a default read/write dependency and a backedge dependency straight forward. An alternative to using a domain expert is to apply topological sorting to the graph to determine an execution order (see Figure 6.20). However, this sometimes leads to multiple schedules of which one has to be selected. This still requires domain knowledge to improve performance and/or numerical stability.

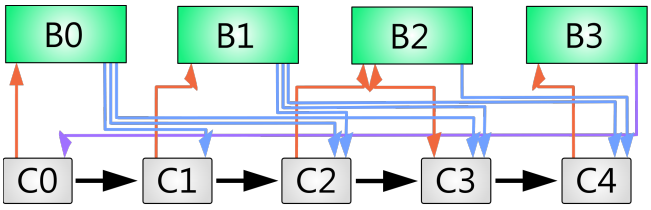

Figure 6.19: A simulation workflow of four components and their read (blue arrows) and write (orange arrows) dependencies to four different buffers (green boxes). The backedge data-dependency is highlighted in purple.

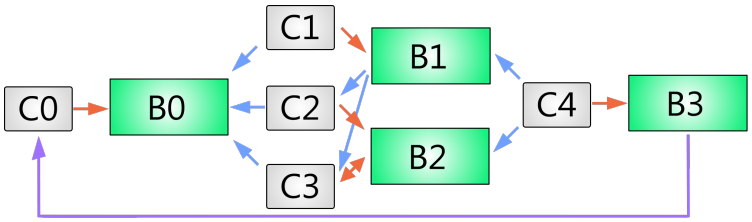

Figure 6.20: A different layout of Figure 6.19 that visualizes the dependency graph without a pre-determined execution schedule [KGK20a]. Topological sorting would also lead to the same component schedule as before C_0, C_1, C_2, C_3, C_4. Furthermore, the backedge in purple makes it easy to distinguish between the first and the last component in the schedule.

The next step after determining the component schedule, is reasoning about the adaptive time-step size computation (see Figure 6.21). Initially, we query all components by asking them to compute their maximum compatible time-step size using the same input state for all components (1). Programmatically, this is done by calling the `ComputeNumSteps` method shown earlier (see also Listing 6.4). In an ideal world, we could use the maximum time step resulting

from this estimation step. Unfortunately, this will not work, as the component consuming data via a backedge would also need to get interpolated information, which is not available. Therefore, the maximum step size is given by the maximum step of size of the component reachable via a backedge dependency in the dependency graph (if there is only one). If this is not the case and there are multiple components reachable via back edges, the minimum step-size over all of these components will define the next maximum step size.

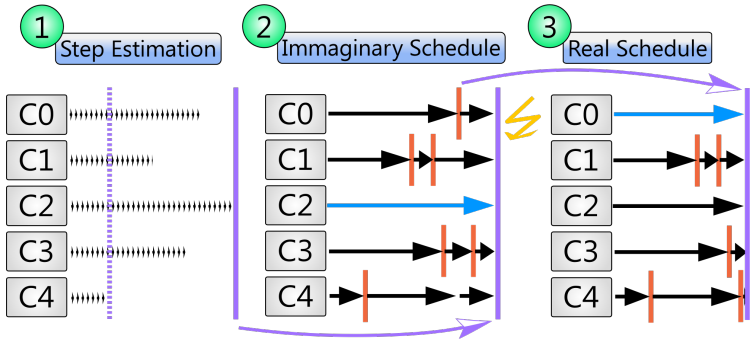

Figure 6.21: Our 3-step method to realize adaptive time stepping in arbitrary domains [KGK20a]. This figure assumes the five components and their associated dependencies from Figure 6.20, as well as their individual potential next time-step values (1, black arrows). The minimum total step size is indicated by the dashed line in purple. An imaginary schedule (that ignores back edges) is shown in (2). Here, component C_2 defines the maximum step size possible and implies several interpolation steps for all other components (orange bars). Due to the backedge data dependency of C_0, this schedule cannot be realized as it requires data from the previous iteration to be interpolated using data that has not been computed. Consequently, the real schedule (3) uses the maximum step size of C_0 to define all interpolation points.

6.3.1 Cached Interpolation Results and Cache Integration

A major disadvantage of our method is the need for on-the-fly interpolation-function evaluations. Depending on the interpolation functionality being used, the additionally imposed computational overhead of re-applying this function on every access can cause performance penalties (see Figure 6.22). Even worse, an increased number of memory accesses (to even fetch target buffer information for interpolation purposes) implies further overhead. A commonly used solution to this kind of problems is the use of shared memory. This solution is also applicable in this scope when caching results of previously interpolated data. In our case, we use a few cached values per thread and assume an access window of threads to be limited to data within a group (see below).

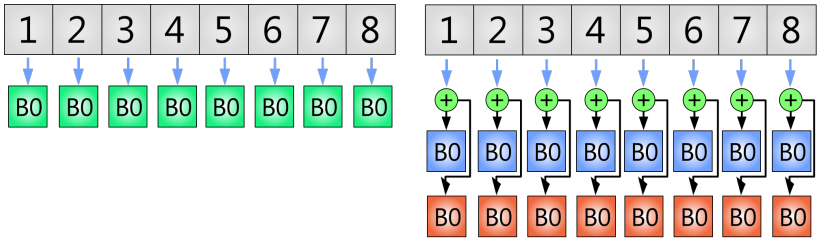

Figure 6.22: Several threads in the scope of a group accessing items in global memory from a buffer (green, left) [KGK20a]. Two loads from a source (orange) and a target buffer (blue) are introduced when using an interpolation function. This requires additional time and resources to compute the interpolation result (indicated via the green plus).

However, the access patterns of components vary in general (see Figure 6.23). This implies that not all accesses can be explicitly cached in shared memory due to its size restriction [KGK20a; NVI23a]. An often seen access pattern in the scope of GPU-aware programming are coherent accesses within components to ensure the highest performance on GPUs anyway (local access window, see Chapter 2). Consequently, this automatically implies that most memory accesses to interpolated data will be cache hits with respect to our introduced caching logic. However, more advanced access patterns cannot be covered with the presented caching approach if their access leave the cached access window. In these cases, more advanced static program analyses are required to perform more sophisticated global program transformations.

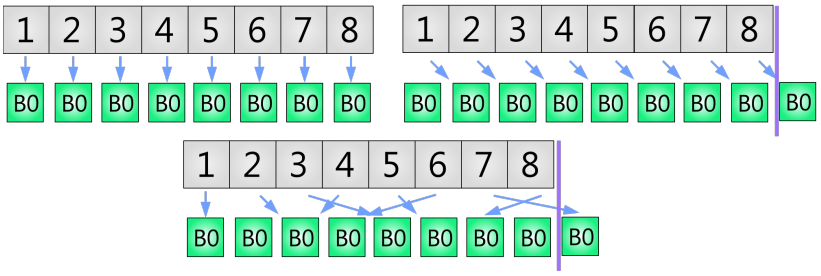

Figure 6.23: The most common coalesced access pattern (top left) ensures that all threads access elements within the cache [KGK20a]. Shifted, but still coalesced, access pattern (top right) ensures that most threads access elements within the cache. Accesses that go beyond the boundary in purple will required further on-the-fly interpolation. Even such an access pattern (top right) can usually be covered by our caching concept due to compile-time program analyses. However, random memory access patterns (below) are not beneficial in some cases, as they tend to access memory locations beyond the purple boundary without being able to statically know this in advance.

An important question in this scope is on how to implement such a caching concept without touching the component implementations during and after development. As demonstrated before (see Listings 6.1 and 6.3), it is possible to define generic component interfaces that do not rely on a specific buffer-/view implementation. In this way, we can provide a specialized cached view as a source buffer by hiding its implementation details. Based on the component schedule and information we get about memory accesses and interpolation points in this schedule, we use this information to automatically generate cached views. Listing 6.6 shows such a sample implementation that has been automatically generated by our code-generation phase. Note that `Type0`, `Type1`, and `Type2` are automatically generated types that represent cached information for two intermediate results of two components. The same applies to the methods `GetValueN` that either fall back to an interpolated result, do the interpolation on-the-fly or just return the non-interpolated value.

Listing 6.6: Cached data view implementation in pseudo-C#/C++ code

```
struct CachedDataView<TInterpolationFunction> : DataView
  where TInterpolationFunction : IInterpolationFunction
{
  // Shared-memory pointers.
  private shared Type0* cachedType0;
  private shared Type1* cachedType1;
  private shared Type2* cachedType2;
  // ...

  private int cacheBaseIndex;

  CachedDataView(
    BufferData sourceBuffers,
    BufferData targetBuffers,
    int index,
    float interpolationFactor,
    // Shared-memory pointers.
    shared Type0* type0Cache,
    shared Type1* type1Cache,
    shared Type2* type2Cache,
    // ...
    // Initialize the basic DataView type to wire the required buffer
        pointers to point to the actual source and target buffers.
    : base(sourceBuffers, targetBuffers)
  {
    // Compute base index of the currently cached data.
    caseBaseIndex = rangeIndex - Group.Index;
    // Load interpolated values from source buffer.
    cachedType0 = TInterpolationFunction.Interpolate(
      sourceBuffers.GetValue0(index),
      targetBuffers.GetValue0(index),
      interpolationFactor);
    // ...
  }

  // Returns the interpolated value0 of Type0.
  Type0 GetValue0(int index)
  {
    // Check whether we have already computed an interpolated value.
    if (index >= cacheBaseIndex & index < cacheBaseIndex + Group.Size)
      return cachedType0[index - cachedBaseIndex];
    // Cache miss.. load values from global memory and use interpolation.
    return TInterpolationFunction.Interpolate(
      sourceBuffers.GetValue0(index),
      targetBuffers.GetValue0(index),
      interpolationFactor);
  }

  // Returns the interpolated value1 of Type1 without using a cache.
  Type1 GetValue1(int index) => TInterpolationFunction.Interpolate(
    sourceBuffers.GetValue0(index),
    targetBuffers.GetValue0(index),
    interpolationFactor);

  // Do not apply interpolation mechanisms to not-affected values.
  Type2 GetValue2(int index) => base.GetValue2(index);

  // ...
}
```

6.3.2 Algorithms

The core part of our approach is the generic algorithm to apply our method to an arbitrary component C_i in the scope of a simulation (see Algorithm 16). It was specially developed for GPUs and to be specialized or automatically generated by a compiler. This can be achieved by either using meta-programming techniques [Kös+14c] or full-featured code-generation phases in one of the compiler backends being used. Specializing or instantiating this algorithm for each component type allows us to realize zero-cost abstractions without worrying about potentially introduced overhead.

In terms of the algorithm itself, we start by providing the maximum number of steps to perform. This number is determined in advance using the concepts described above. In all cases we begin a component execution by allocating shared memory for all intermediate values to be cached depending on information derived from the dependency graph. The next step is to cover all values in the range of our component C_i.

However, we have to use the padded value range of C_i (to be a multiple of the grid-stride step size) to avoid divergent control flow, and thus, unintended side effects like deadlocks on certain architectures. Then, we enter the important inner-most loop to do the actual adaptive time stepping and start by initialize our shared-memory caching view (see Listing 6.6). The following steps realize the upcoming step-size calculation via hierarchical group-wide reductions to compute the upcoming step-size minimum (lines 15–25). Finally, we must check whether we are out-of-range with respect to our actual value range and evaluate the component C_i using the next time step before we update the current step information and perform cleanup copy operations to ensure a valid state for the next component.

Implementation Details

We used C# to realize our adaptive time-stepping method and leveraged the ILGPU [Kös23] GPGPU-JIT compiler for GPU workloads. Each component was realized in C# via a generic structure that implements the previously presented component interface (after translating it into the C# world). Due to the fact that C# is compiled to an intermediate representation [Lid02], it was possible to inspect the emitted .Net assembly format. This gave us the ability to inspect component dependencies in order to resolve the dependency graph based on meta-data information stored within the assemblies and implemented disassembly utilities [Kös23]. We then generated specialized GPU kernels (based on Algorithm 16) automatically for each component while also creating reduction kernels to compute the maximum number of steps per iteration. This also involved the emission of specialized shared-memory-based views (based on Listing 6.6) to speed up interpolated lookups. In order to implement all reductions, we relied on atomic operations and efficient warp reductions [NVI14; NVI23a] (see also Algorithm 15).

Algorithm 16: Our time-step adaptive simulation algorithm for an arbitrary component C_i [KGK20a]

```
   Input: max#Steps, sourceBuffer, orgSourceBuffer, targetBuffer
   /* All intermediate values are cached in shared mem     */
 1 memCache0 = shared memory Type0[group size];
   /* ...                                                  */
 2 stepSize := shared memory int[1];
   /* Iterate over the padded value range of C_i to avoid
      divergent control flow wth respect to all other
      threads in the group                                 */
 3 for j := global index to pad(range(C_i)) step by (grid size * group size)
    do
 4     tempSource, tempTarget := sourceBuffer, targetBuffer;
 5     #iterations := 0;
 6     for step := 0 to max#Steps; do
 7         view := new CachedDataView<Interpolation Function>(
 8             tempSource, tempTarget, orgSourceBuffer, j,
 9             max#Steps / (step + 1) as float,
               /* All shared memory buffers...             */
10             memCache0, ...);
           /* Initialize the step size and wait for all
              threads in the group to reach the barrier    */
11         if thread index = 0 then
12             stepSize := max#Steps - step;
13         end
14         group barrier;
           /* Check whether we have exceeded our value range
              */
15         currentStepSize := max(int);
           /* Compute next common step size for all group
              threads                                      */
16         if j < range(C_i) then
17             currentStepSize := C_i.ComputeNumSteps(view);
               /* Determine the next step size by reduction */
18             reducedStepSize := warp reduce min(currentStepSize);
19             if lane index = 0 then
20                 atomic min stepSize, reducedStepSize;
21             end
22         end
23         group barrier;
24         actualStepSize := stepSize;
           /* Apply component with next common step size   */
25         if j < range(C_i) then
26             C_i.Evaluate(view, actualStepSize);
27         end
           /* Increment step index and iteration count     */
28         step := step + actualStepSize;
29         #iterations := #iterations + 1;
30         Swap tempSource, tempTarget;
31         group barrier;
32     end
33     if #iterations mod 2 = 0 then
34         Copy all information from the sourceBuffer to the target buffer
35     end
36 end
```

6.3.3 Performance Evaluation

In order to have a reasonable performance evaluation for our adaptation method presented above, we designed two application scenarios relying on the basic principles of particle-based simulations. Both scenarios were modeled using high-level dependency graphs providing detailed information about the components being used and their dependencies between them. In analogy to the evaluation of our thread-compaction based method to model interpreters involving multiple states, we also avoided hard-to-understand and closed-source benchmarks. We also leveraged matrix-matrix multiplications (see Section 6.2.3) in each component to avoid additional memory accesses introducing further side effects while making sure to add reasonable workload per thread and component. However, we did not consider simulation deviations, as these must be investigated separately for each domain and use case, which ultimately may require custom interpolation functions. We ran our benchmarks on two GPUs from NVIDIA featuring different compute capabilities [NVI23a]: GeForce GTX 980 Ti and GeForce GTX 1080 Ti [KGK20a][3]. We considered as a single performance measure the mean execution time of 100 simulation runs. In each run, we performed at most 100 simulation steps which corresponded to running the simulation with a conceptual step size of 1 per step[4].

To come up with benchmarks actually reflecting real-world uses cases from the particle domain, we added specific memory accesses to neighboring items in memory. When neighborhood accesses were required to model such a use case, we always simulated neighboring particle accesses by 9 IO operations, which usually arise in SPH-based simulations [MM13; GKK19; GKK20]. Note that this number was chosen to cover 2D grid-based access patterns which access all 8 neighboring cells in addition to their current cell. Although we did not evaluate numerical deviations, we did evaluate the use of more and less expensive interpolation functions based on linear and cubic spline interpolation.

The number of particles has a tremendous impact on the runtime of such simulations and is referred to as the value range ($range(C_i)$ of each component) in the scope of the evaluation. Hence, we used two value ranges (16384 and 65536) to evaluate scaling behavior and to ensure a reasonable workload for our evaluation GPUs. Note that the ranges were used for all components to ensure a proper workload distribution. This avoided focussing on measuring certain performance characteristics of the underlying simulation model rather than the adaptation algorithms. In addition, we always accessed all items in the value range (all particles) using optimized coalesced memory accesses.

[3] Note that both GPUs were state-of-the-art at the time of the publication. Therefore, we retained these measurements to present the original results. However, it is of interest to reevaluate our method on more recent GPUs in future work (see Chapter 14).

[4] Note that using an adaptation scheme reduces the number of simulation steps intentionally. This results in less simulation steps for different adaptation schemes.

As explained before, the number of simulation steps is the most important bottleneck to be addressed with our adaptive time-stepping concept. We continuously varied the number of steps per component by drawing a sample from a uniform random distribution for each component. This ensured that we are close to actual real-world applications which adjusted the number of steps per component continuously throughout the run of a simulation to compensate artifacts and/or to ensure stability [Mül08; MM13; KK18; KGK20a]. To be inspired by such applications we sampled from the interval $[1, \ldots, 3]$. This avoided optimistic assumptions about artificially increased potential speedups that could be achieved with our method. Note that this also mimics common use cases in which we may do larger steps from time to time.

The performance evaluation compared different methods to each other[5]:

- The *traditional adaptive* approach (referred to as *Trad. Adaptive.* in all diagrams) using a simple adaptive time-stepping method based on previous research (see also Section 6.1 and Figure 6.4),
- our adaptive time-stepping method without shared-memory caches (referred to as *HIPnC*, HIP no cache), and
- our adaptive time-stepping method using shared-memory caches (referred to as *HIP*).

Moreover, we also combined both of our approach configurations (with and without shared memory) with a linear and a cubic-spline (*spline2*) interpolation function. For instance, combining our shared-memory enabled method *HIP* with a *spline2* function is referred to as *HIP Spline2*.

The first evaluation scenario was a simulation inspired by N-body gravity simulations (see Section 3.1) that involved three components (see Figure 6.24). Different components modeled specific parts and access-pattern characteristics of such a simulation.

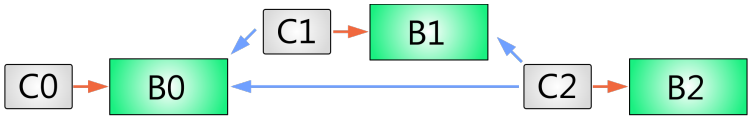

Figure 6.24: Gravity-simulation like workflow that involved three components [KGK20a]. From a high-level point of view, C_0 computed location and particle-specific data that was passed to C_1 which computed neighbor-based gravity forces. C_2 took this intermediate information and applied the resulting forces to all particles.

Speedups comparing the different adaptation methods for this evaluation domain are shown in Figure 6.25 and Figure 6.26. The latter one used 65536 particles to enable the direct comparison of a smaller simulation domain to a larger one.

[5] All speedups shown in the diagrams were measured in relation to the non-adaptive simulation that served as the baseline

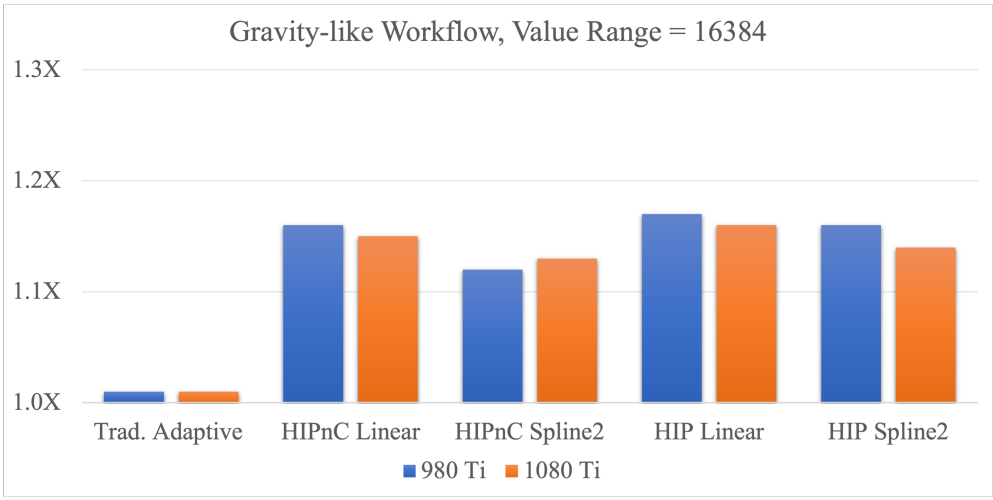

Figure 6.25: Speedups comparing the non-adaptive gravity-simulation like implementation to the traditional adaptation approach and to our method using different interpolation kernels (higher is better, based on performance numbers published in [KGK20a]). Note that these speedups were measured using 16384 particles as the value range for our components.

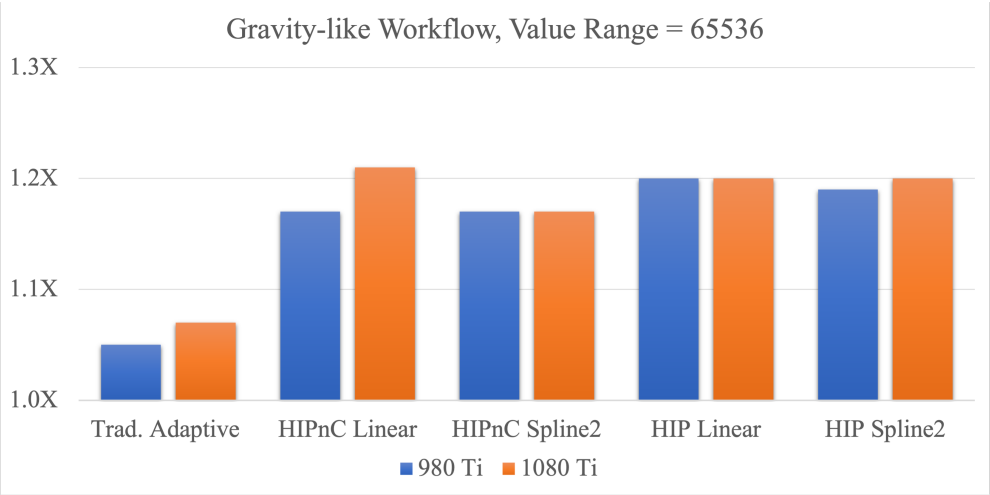

Figure 6.26: Speedups comparing the non-adaptive gravity-simulation like implementation to the traditional adaptation approach and to our method using different interpolation kernels (higher is better, based on performance numbers published in [KGK20a]). Note that these speedups were measured using 65536 particles as the value range for our components.

The execution time of 100 non-adaptive simulation steps was ≈ 136ms on the GTX 980 Ti and ≈ 83ms on the GTX 1080 Ti [KGK20a]. As shown in the diagrams, the traditional approach did not yield to any significant performance improvement when using a value range of 16384 (Figure 6.25). This changed slightly when the number of particles was increased on both GPUs, resulting in increased speed between 1.05× on the GTX 980 Ti and 1.07× on the GTX 1080 Ti. Compared to the traditional method, our adaptation methods achieved substantial speedups of 1.12× to 1.16× without caches and 1.14× to 1.17× with caches enabled. The negligible performance improvements using caching were related to the additional overhead managing the cache that nearly outperformed its benefit of reducing memory transfers. Related to that are the performance degradations using the cubic spline kernel which created more computational overhead on each access that caused these performance improvements to be less than the ones using linear interpolation.

These effects changed when the value ranges were increased by 4×, which yielded more substantial speedups on both evaluation GPUs (Figure 6.26). Overall, we maintained a good speedup compared to the non-adaptive simulation ranging from 1.17× to 1.21×. However, the performance improvement over the traditional adaptation method decreased to 0.7× in the worst case on the GTX 1080 Ti (in comparison to the baseline measurements). A combined speedup comparison of all measurements from the gravity domain is shown in Figure 6.27. With the exception of one outlier measurement without caching in combination with the 65536 particles on the GTX 1080 Ti, caching always improved the performance by additional 0.3×. This is caused by the negligible overhead of linear interpolation in comparison to the number of global memory accesses.

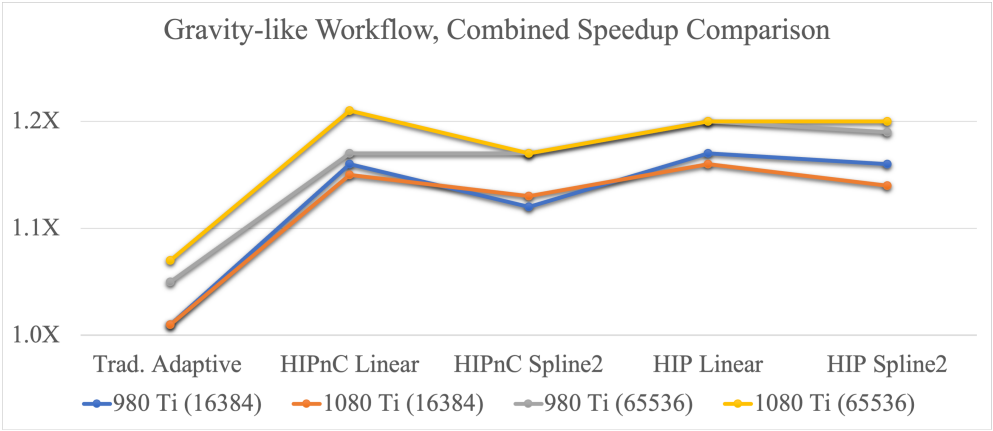

Figure 6.27: Combined speedups from Figure 6.25 and Figure 6.26 to directly compare the achieved speedups using two different value ranges of 16384 and 65536 (higher is better, based on performance numbers published in [KGK20a]).

The second evaluation scenario was a simulation inspired by position-based dynamics/fluids simulations (see Section 5.1) that involved seven components (see Figure 6.28). As before, various components represented specific features of PDB/PBF simulations, such as iterating over neighboring particles and aggregation of intermediate information required to perform collision detection and response mechanics, for example. Speedups when comparing the different adaptation methods for this evaluation scenario are shown in Figure 6.29 and Figure 6.30.

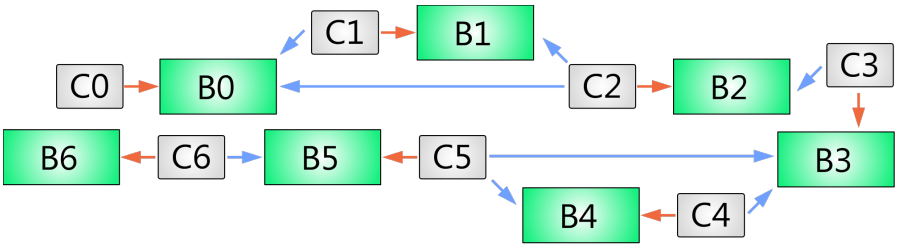

Figure 6.28: PBD/PBF-simulation like workflow that used seven components [KGK20a]. Similar to Figure 6.24, C_0 prepared simulation- and situation-specific information for each particle. This information was passed to C_1 which accumulated neighbor information for collision detection/response steps realized in C_2. Similar to the approach shown in Section 5.1.3, C_3 and C_4 relied on fluid-dynamics calculations derived from SPH-based neighbor processing. Finally, C_5 implemented further constraint propagation updates for non-fluid dynamics objects and C_6 commited all pending changes in terms of velocity and position updates.

The execution time of 100 non-adaptive simulation steps was $\approx$ 327ms on the GTX 980 Ti and $\approx$ 200ms on the GTX 1080 Ti [KGK20a]. As shown in both figures, the traditional approach resulted in speedups ranging from 0.92$\times$ to 0.95$\times$ – slowdowns effectively. This was caused by the increased number of components that often resulted in step sizes of 1 while adding overhead to determine next step sizes to perform. When using 16384 particles in the simulation, we measured performance improvements ranging from $\approx$ 1.1$\times$ to $\approx$ 1.18$\times$ on both evaluation GPUs. The older GTX 980 Ti performed slightly better when caching was enabled, whereas the GTX 1080 Ti did not benefit from explicit caching. The reason for this was the improved memory throughput and the considerably improved computing performance of the GTX 1080 Ti compared to the GTX 980 Ti.

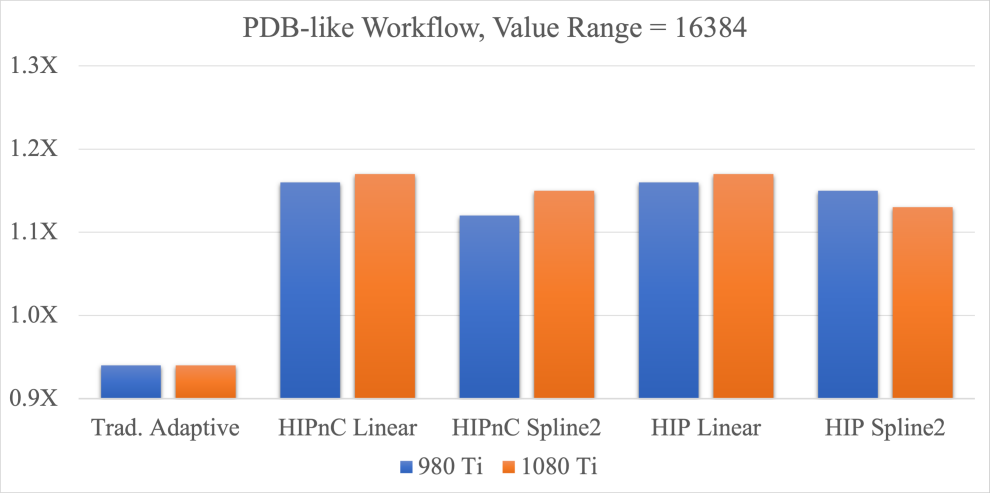

Figure 6.29: Speedups comparing the non-adaptive PDB-simulation like implementation to the traditional adaptation approach and to our method using different interpolation kernels (higher is better, based on performance numbers published in [KGK20a]). Note that these speedups were measured using 16384 particles as the value range for our components.

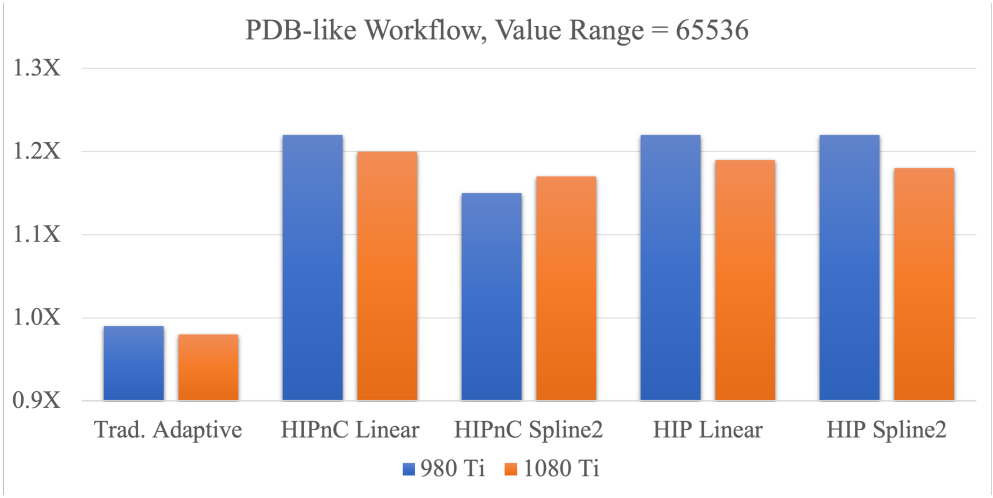

Figure 6.30: Speedups comparing the non-adaptive PDB-simulation like implementation to the traditional adaptation approach and to our method using different interpolation kernels (higher is better, based on performance numbers published in [KGK20a]). Note that these speedups were measured using 65536 particles as the value range for our components.

By increasing the value range to 65536 elements, we were able to reliably measure speedups from 1.15× (on the GTX 980 Ti) to 1.21× on both GPUs. This significantly outperformed the traditional adaptation method and the non-adaptive base-line simulation. However, enabling caching was of little benefit as the interpolation features were not expensive enough to outweigh the cache management overhead on both GPUs.

As for the first evaluation scenario, a combined speedup comparison of all measurements from the current domain is shown in Figure 6.31. Although we achieved substantial performance improvements using all interpolation functions, using the cubic spline interpolation function always came at a certain price. This is particularly evident in the PBF-like domain, as more components require more interpolation operations, which is clearly visible in the graphs.

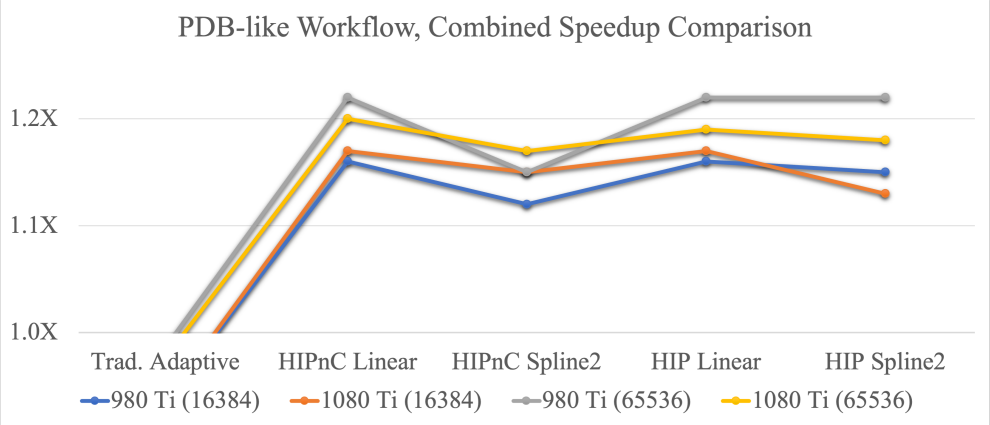

Figure 6.31: Combined speedups from Figure 6.29 and Figure 6.30 to directly compare the achieved speedups using two different value ranges of 16384 and 65536 (higher is better, based on performance numbers published in [KGK20a]).

CHAPTER 7

CONCLUSION

The first two sections, 5.1 and 5.2, covered contributions in the field of particle-based simulations and selection processes. Our APBF method to adaptively adjust the number of solver iterations for PBD/PBF-based simulations achieved speedups ranging from 1.4× to 1.7×. We introduced two adaptation models to tweak the number of iterations in a fine-grained way without causing instability of the underlying simulation. Regarding the rendered quality of the simulations based on our adaptation models, we were able to maintain a high visual quality that did not suffer from artifacts caused by lower-quality PBF simulations. Moreover, the method seamlessly integrates with common PBD/PBF solvers, making this contribution directly applicable to real-world applications.

Similarly, our method for realizing particle selections in $O(n \cdot k)$ performed very well in our evaluation scenarios, where n stands for the number of particles and k for the maximum number of neighbors a particle can possibly have [KK18]. This was achieved by decoupling several steps of the volume analyses from the dataset and performing them in screen space instead. Our measurements showed that we obtained excellent results in terms of selection quality and are on par with methods from related work. Also particularly important is the negligible memory consumption and the impressive runtime of our method. We were able to perform more than 30 selections per second, allowing us to process changing datasets (e.g., based on simulations) in real time. Most importantly, scalability was greatly improved since our algorithms do not work on uniform grids, and we improved performance by at least an order of magnitude over the most similar method, which assumed precomputed information. Disabling precomputed density information for our competitor increases our speedup to at least two orders of magnitude. Like APBF, our method can be directly integrated into visualization systems because our method is self-contained in terms of algorithms and does not require specific data structures.

Sections 6.2 and 6.3 focused on generic parallel simulations built from multiple components in the context of many optimization states processed in parallel. Each state represents data needed by a higher-level optimization to explore the search space, while the components process all states (and the data elements they contain). Our proposed method from Section 6.2, which is based on thread compaction, leverages domain knowledge about the way components in such systems are usually constructed according to the concept of inference rules. To take full advantage of the algorithm, we presented a new, specially-optimized memory layout for parallel processing of multiple states to ensure coherent memory accesses. Overall, our evaluation showed that our proposed memory layout outperformed conventional approaches by a factor of 1.7× to 3.3×. The layout itself outperformed prominent and often-chosen layouts by a factor or 1.1× to 1.25× when our algorithm was used. Since our algorithm is a generic execution algorithm for simulations using components, our concept can be easily integrated into existing optimization systems without the need to adapt the component logic. In analogy to the algorithm itself, the data layout can also be taken advantage of without touching the component implementations. However, this assumes a certain design of such a framework or system, which allows changing data layouts using view abstractions (see Section 6.1).

In Section 6.3, we contributed a novel method to realize adaptive time stepping for generic component-based simulations. We leverage interpolation functions to calculate intermediate values within optimization states, which helps relax time-step-size restrictions. Depending on the interpolation function used, it is also possible to use our method with custom-designed caches that seamlessly integrate with component designs supporting generic memory layouts. The general idea of our approach is to assume larger time-step sizes based on an optimistic assumption, as opposed to pessimistic assumptions typically made due to domain-specific requirements in certain research fields. However, since we focus on general-purpose simulations operating on optimization states, we can benefit from less stringent constraints and solving the actual adaptive time-stepping problem at a higher level. In benchmarks, we measured speedups of 1.17× to 1.21× on our conservatively chosen evaluation scenarios compared to traditional adaptive time-stepping approaches.

Answering RQ1 All presented approaches significantly improved performance compared to current state-of-the-art algorithms. We contributed to domain-specific and domain-independent fields. Here, we mainly targeted interaction methods based on simulations, interactive simulations, and simulations in the scope of optimization systems. In summary, this contributes to answering $RQ1$, as we have shown how we can achieve significant speedups over existing methods through shorter execution times, lower memory consumption, and better scalability (see also Chapter 13).

Part II

Heuristic Optimization

CHAPTER 8

INTRODUCTION

Part 1 introduced the motivation and fundamentals of modeling massively-parallel simulations and presented several methods for improving the performance of simulations on GPUs. As mentioned in Section 1.2, Part II focuses on performance improvements of heuristically driven optimization systems. Reconsider a high-level search tree of a simulation-based optimizer instance from the preface (see Figure 8.1). Part 1 allows us to benefit from considerable performance improvements in the simulation-logic part of such an optimization system. This in turn enables us to evolve states in terms of simulating actions and their behavior on the observed states. Unfortunately, this does not cover the actual state-generation phase, which is responsible for scheduling actions to be applied to different states in order to observe their behavior over time.

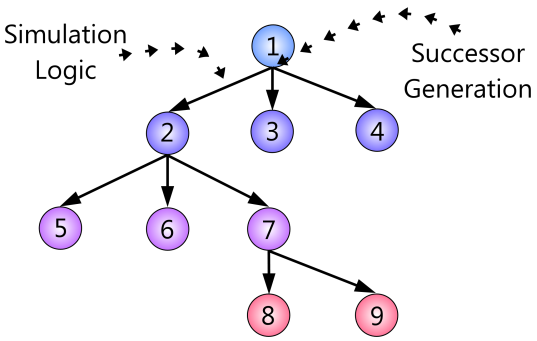

Figure 8.1: Conceptual search-tree-based method [KGK19c] (see Figure 1.2).

However, this phase is covered in this part along with more general high-level methods that can be used to efficiently remember certain states without exploding memory usage. It is particularly important for solving large-scale problems involving many optimization states per level and a large tree depth in general. Although there are sophisticated pruning algorithms and concepts

in the field of large-scale optimization, this is still not sufficient to eliminate most of states, as they still need to be expanded, explored, and evaluated. In such cases, our method(s) help to reduce memory consumption by orders of magnitude (see Chapter 10).

A very related and challenging task in this field is the expansion strategy of states themselves (see Figure 8.1, Successor Generation). In this context, the task is to (efficiently) explore the directly reachable successor states of a given optimization state (see Figure 8.2). This problem has been tackled by many researchers in the past using complex and domain-specific algorithms. However, realizing a solution that can be efficiently executed on GPUs is even more difficult. Although related work also addressed many challenges before we started our research, it turned out that all methods were problem and/or domain specific. Solving this problem in a problem-domain independent manner is even harder. In addition, the most important aspect we wanted to ensure was that data did not leave the GPU device during optimization for performance reasons. As far as we know, most of these requirements have not been fulfilled in previous work. In order to satisfy all the requirements, we invented a novel method that realizes generic successor-generation on GPUs for heuristic optimization problems in Section 10.2.

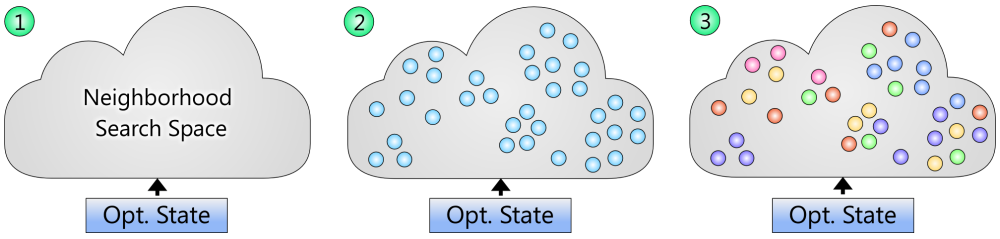

Figure 8.2: Conceptual exploration of successor states in the neighborhood of a given state [KGK19a]. Conceptually, the neighboring search space is often tremendously large and often even difficult to explore (1). The first challenge is to efficiently enumerate potential successor states that are directly reachable in terms of required simulation steps (2). Once neighboring states have been identified, they are prioritized to guide the exploratory search (3).

8.1 Contributions

After discussing related work from the domain of heuristic optimization in Chapter 9, Part II (see Chapter 10) of this thesis makes the following important contributions:

- Section 10.1 presents a generic method to maintain optimization states in accelerator memory and shows how to explore different (heuristically controlled) paths in the scope of heuristic optimization trees. This approach specifically focuses on the parallel management of a large number of states, where the different states can be in different levels (different exploration tree depths). Moreover, we also handle and solve a special kind of memory-management problems that usually arise in large-scale optimization search trees. This allows us to overcome scalability issues on large-scale problems by significantly reducing the amount of memory required.

- Section 10.2 describes a domain-independent method for generating successor states during search-based exploration on GPUs without involving the CPU in any way. This particularly affects evolutionary, meta-heuristic-based, and genetic optimization systems. All of these approaches rely on a successor generation function, also known as neighborhood or neighborhood-exploration function. Furthermore, it allows using priority-sampling based exploration heuristics and can be seamlessly integrated into any heuristic optimization system.

Both approaches overcome known limitations of heuristic and exploration-based optimization systems in terms of memory consumption, runtime performance, and thus scalability and applicability to large-scale problems. In analogy to Part 1, all algorithms presented in this part have been designed with GPUs in mind and can be (re-)implemented in any GPGPU-capable programming language. Figures, code listing and pseudo-code snippets are discussed in detail while providing implementation-relevant hints and annotations to facilitate the integration into real-world applications.

8.2 Publications

The following list summarizes all contribution-relevant publications of this part of the thesis. Furthermore, contributions of the contributing author (*CA*) and all other authors (*CoA*) are explicitly listed. This helps to clearly separate own work that may be used in the dissertation from other contributions.

[KGK20b] Marcel Köster et al. "Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction." In: *International Journal of Parallel Programming* (2020)

CA Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking

CoA Feedback on the paper, paper refinement, implementation and benchmarking

[KGK19a]⊛ Marcel Köster et al. "FANG: Fast and Efficient Successor-State Generation for Heuristic Optimization on GPUs." In: *19th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2019)*. Springer, 2019

CA Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking

CoA Feedback on the paper, paper refinement, implementation and benchmarking

[KGK19c]⊛ Marcel Köster et al. "Parallel Tracking and Reconstruction of States in Heuristic Optimization Systems on GPUs." In: *Parallel and Distributed Computing, Applications and Technologies (PDCAT-2019)*. IEEE, 2019

CA Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking

CoA Feedback on the paper, paper refinement, implementation and benchmarking

[KGK19b]	Marcel Köster et al. *Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction.* 12th International Symposium on High-Level Parallel Programming and Applications (HLPP-2019). 2019
CA	Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking
CoA	Feedback on the paper, paper refinement, implementation and benchmarking
[KK16]	Marcel Köster and Antonio Krüger. "Adaptive Position-Based Fluids: Improving Performance of Fluid Simulations for Real-Time Applications." In: *International Journal of Computer Graphics & Animation* (2016)
CA	Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking
CoA	Feedback on the paper, paper refinement

CHAPTER 9

RELATED WORK

This chapter summarizes related work in the area of maintaining multiple (intermediate) optimization states in memory and exploring the search space via neighborhood enumeration. It aims to provide a high-level overview of related approaches and algorithms that are most similar to our contributions, as well as their limitations. Although concepts for maintaining multiple states on GPUs are widespread and commonly used, most publications focus on custom use cases or domain-specific solutions [KGK19a; KGK19c]. Similarly, neighborhood exploration has been well researched in lots of domains while focusing on parallel GPU processing [KGK19a]. The first section covers publications related to all our contributions in this part. However, Section 9.2 summarizes related work that is most similar to our methods from Section 10.1, while the work discussed in Section 9.3 is related to our successor-state generation from Section 10.2. Each section also includes paragraphs summarizing key aspects of related work, as well as the main differences to our approaches.

Please note that this list of publications cannot be exhaustive by definition, as we have made contributions to fields that touch on a wide range of domains.

9.1 Parallel State Tracking and Neighborhood Exploration

Directly related to our methods is work on a variety of different frameworks for realizing parallel constraint solving on GPUs. In this domain, it is always required to remember and keep track of different partial solution states before continuing the expansion process into a certain direction. Prominent foundations had been built by Campeotto et al. [Cam+14b]. The authors used a hybrid approach to maintain most of the information about the intermediate optimization state in CPU. To realize accelerated propagation steps, they transferred this data to the accelerator. Here, the CPU part of their system

made the decision which variable (sub)sets and assignments to explore further. The GPU accelerated parts then processed various possibilities and made the results of this phase accessible to the CPU-side of the application via memory copies. Hence, their actual solver ran on the CPU and used the GPU for accelerated evaluation of different potentially interesting variable combinations while using transfers between CPU and GPU all time time. This also means that their neighborhood exploration used the GPU for rating different possibilities, whereas the actual successor-state generation happened on the CPU. In contrast to them, we track objects exclusively on the GPU and do not perform copy operations between the different phases to minimize transfer overhead (see Section 10.1 and Section 10.2). We also store densely packed information on the CPU side when needed, for instance, running out of GPU memory during optimization. However, the size of the memory buffers to be transferred is orders of magnitude smaller than the buffers used by related work. This is due to the fact that we used highly compressed information from which states can be reconstructed if needed (see Section 9.2 and Section 10.1).

Campeotto et al. [Cam+14a] published a follow up paper that revealed more detailed information about the internals of their neighborhood exploration method. As mentioned before, they still focused on the CPU when it came to the set of variables to explore. However, they used so called *local search strategies* [Cam+14a] to determine these sets. Although the authors also proposed certain approaches that worked well in their domain, the high-level method of using local search-space exploration is highly similar to our idea (see Section 10.2). However, we realize the whole neighborhood exploration functionality on the GPU with our newly presented approach. The interested reader may refer to work by Focacci et al. [FLL04] for more information about theory and application of constraint satisfaction and local search-space exploration in this domain.

Munawar et al. [Mun+09] published an efficient GPU-based algorithm including local neighborhood search for MAX-SAT problems. They introduced the concept of a virtual 4D grid designed for their genetic algorithm. The underlying idea is that two dimensions are used for different neighbor assignment possibilities for each individual element inside a genetic algorithm population and two dimensions for each population itself. While it is not as important to understand the detailed mechanics of their proposal, it is important to know that their 4D grid, which includes two 2D exploration dimensions, can be represented directly on the GPU for parallel processing. This is closely related to our approach as we also realize neighborhood exploration efficiently on the GPU without CPU-based data transfers. However, our algorithm is not bound to such a MAX-SAT optimization problem and is not restricted to such an encoding of different possibilities. Instead, we can work with arbitrary local heuristics, which may also be given in the form of neural networks (see Section 10.2).

A related approach that implicitly borrows from the concepts presented earlier is the paper published by Abdelkafi et al. [ACK13]. They maintained states on the CPU and copied necessary information to the accelerator for improved parallel processing performance during exploration phases. The authors used OpenCL to process individual variable assignments in parallel on the GPU but evaluated each neighbor rating sequentially afterwards on the CPU side. To make their OpenCL-based approach work for their evaluation domains (the *knapsack* and the *traveling salesman (TSP)* problems), they used domain-specific data structures to realize neighborhood exploration. Comparing their contributions to ours, Abdelkafi et al. were very limited in the sophisticated way they had to add support for different variable assignment strategies. They also suffered again from the state tracking problem by keeping most data on both the CPU and GPU.

Also a very prominent search algorithm is *tabu search* which was also investigated by Luong et al. [Luo+10]. This is a meta-heuristic search method that they parallelized on the GPU, which specifically involved exploring local neighborhoods. The contributing author also published more generic papers on local search-space exploration [LMT10a; LMT10b]. They ere interested in the influences of certain decisions within the optimization problem to be solved. Talking about the problem domain itself, the authors aimed at binary decision problems while leveraging *Hamming distances* [Luo+10]. Unfortunately, these approaches are not immediately applicable to arbitrary domains because all methods make several simplifying assumptions. Solving this problem from a general standpoint, a much more sophisticated approach to neighborhood exploration is needed, as opposed to the method(s) they used.

Lam et al. [LTL13] used a related idea while shifting their focus to *simulated annealing*. They purely focused on multi-core architectures in their paper. As Abdelkafi et al., Lam et al. chose TSPs and contributed possibilities to use CPUs and GPUs in their setting. Regrettably, they also leveraged specific mathematical properties of their optimization problem they wanted to solve without generalizing their methods.

Summary and Main Differences to Our Work Related work proposed hybrid approaches to balance CPU and GPU workloads while leveraging computational aspects of both. However, most related work ran the actual solver on the CPU while using GPU acceleration for immediate neighborhood exploration in terms of rating states. Alternatively, they maintained copies in CPU and GPU memory while benefiting from domain knowledge about the shape and type of the optimization problem to solve. This allowed them to integrate domain-specific concepts about neighborhood exploration or rating of states to exploit parallelism and enable state tracking. However, addressing this problem generically for arbitrary domains in a massively parallel and highly efficient way remained unsolved and is being addresses by our methods.

Key difference to related work is that we exclusively materialize states in GPU memory while using densely compressed information to remember states from previous solver iterations. In case we need to fetch information back to the CPU, we only need to transfer tiny amounts of compressed state information as we do not need to materialize any states on CPUs for processing. Furthermore, our approaches works completely domain independent as it does not rely on domain knowledge of the optimization problem being solved. This also includes our method to accelerate neighborhood exploration on GPUs allowing the integration of arbitrary heuristics.

9.2 Parallel State Tracking

A commonly researched and comprehensive method for optimization is *Monte Carlo tree search* (*MCTS*) [Bro+12]. Performance of this approach has been continuously improved over the past years while also porting it to a huge variety of different architectures [CWH08; RS10; XMM18]. Powley et al. [PCW17] had shown a concept around MCTS that allowed them to reuse already allocated states in memory that are not required anymore during the search process. In order to realize that, they used graph-based data structures using pointers to different memory addresses, which is in no way designed for GPUs [NVI23a]. The method presented by Zhou and Zeng [ZZ15] made a similar contribution while improving algorithms such as $A^\star$-based path finding. Reusing pieces of memory is also part of our strategy. However, we rely on pre-allocated memory buffers and reconstruct states on-the-fly when needed, which is far superior to their approach.

In sum, most publications maintain their intermediate data structures (state information) in CPU memory. When they use GPUs to improve the performance of certain tasks, they copy the necessary bits and pieces to the device, perform the computationally intensive operations, and fetch the result back to CPU memory. However, compared to our method, they must keep the actual state information in memory. We use the concept of reconstructing states on-the-fly when needed from densely packed and highly compressed history information of visited states. This allows us to scale significantly better than the other methods discussed in this section.

Rashid and Tao [RT17; RT18] mainly influenced our conceptual idea of state reconstruction. They introduced the idea of a *neighborhood fitness structure* [RT18] information exchange between the GPU and CPU parts of their solver. The authors used this data structure to swap neighborhood information based on variable assignments and state-dependent values. The main difference to our method is that we do not need to transfer detailed information to the GPU for our on-demand state reconstruction. In contrast to them, we just need to transfer a few bytes per state while performing the actually expensive reconstruction in a highly efficient way on the GPU.

Summary and Main Differences to Our Work Focusing solely on the aspect of state tracking, related work addressed this problem in the scope of several methods (like MCTS). Reducing memory consumption was also actively researched in the past. In this scope, related papers mainly targeted reusing already allocated memory buffers during optimization while using domain-specific information about the problem or the solver method being used. However, most publications keep state information in CPU memory and use copy necessary state information to GPU memory for enabling massively parallel processing. This general technique requires large amounts of memory for large-scale problems, which limits their scalability.

The method most similar to our approach uses a fine-grained data structure to exchange information between CPU and GPU to reduce the amount of data being copied. Our approach is conceptually based on their idea, but uses a novel way to compress states and reconstruct them when needed on-the-fly on the GPU. Using our reconstruction technique, we can reduce memory consumption to a few bytes per optimization state in general on arbitrary problems.

9.3 Neighborhood Exploration

Especially relevant for us is the parallelized search-based optimization by Novoa et al. [NQC15] for solving *quadratic assignment problems* using GPU acceleration. In their GPU kernels, which implemented the actual search space exploration, they permuted various variable assignment possibilities based on thread indices within the scheduled thread grid. In order to generate meaningful assignments, they leveraged a *binary decision structure* helping them to map different possibility indices to variable assignments. Naturally, they differentiated between two values for each variable in their domain, namely 0 and 1. This restriction allowed them to considerably simplify their assignment process. Compared to our contributing, we are not conceptually constrained by an arbitrary number of variable assignment possibilities (see Section 10.2). Moreover, we also do not make use of any domain-specific knowledge or assumptions. In order to make our approach scalable and applicable to arbitrary domains, we propose an entirely new way of creating successors on the GPU.

There had been different follow up works based on publications by Luong et al. [LMT10a] that are related to our method. For instance, Ghorpade and Kamalapur [GK14] proposed a method based on Luong et al.'s concept of addressing specific TSP problems via iterating over all neighbors in parallel. They introduced custom data structures and assignment strategies to generate successors based on the possible assignment candidates. Similarly, Melab et al. [Mel+11] presented a whole framework to realize parallel exploration of neighbor states on GPUs. However, the conceptual mechanics of Luong et al.'s method remained the same in terms of the need to transfer information to the CPU side of the application.

Rashid and Tao [RT18] discussed different concepts of neighborhood search, as well as their potentials and challenges in terms of realization and adaptivity. In addition, they presented GPU-based algorithms to implement an optimization system using different kinds of meta heuristics. Unfortunately, they did not go into detail in terms of their actual neighborhood exploration because they assumed an already existing neighborhood exploration function ($N(x)$, see Section 10.2). In contrast to them, we actually contribute a generic realization of an arbitrary neighborhood exploration function without leveraging domain knowledge of a particular domain.

For detailed information about different meta heuristics and meta-heuristic driven concepts, the interested reader may refer to the work by Talbi [Tal09].

Summary and Main Differences to Our Work Recently published papers have utilized GPUs for neighborhood exploration, with the CPU handling general state information maintenance. Related work often assumes domain knowledge about specific problem types or incorporates a given neighbor exploration function into their methods. However, these assumptions limit the applicability of finding a general solution to build GPU-driven optimizers minimizing or even avoiding the need for CPU intervention. Assuming an existing neighborhood rating function simplifies matters significantly and places the burden of parallelizing local assignment heuristics properly on researchers and users.

In contrast to these considerations, our approach provides a generic foundation that allows for seamless integration of arbitrary local heuristics within a GPU-first algorithmic environment. With our method, there is no need for CPU involvement during optimization processes and we do not assume specific characteristics of the heuristics being employed. Furthermore, our approach can significantly improve performance over CPU-GPU methods that rely on data exchange.

CHAPTER 10

IMPROVING PERFORMANCE OF HEURISTIC OPTIMIZATION

This chapter presents contributions to considerably improve performance and to reduce the memory footprint of heuristic optimization algorithms on GPUs. The most important conceptual building block in this scope is internal search trees, which are built either implicitly or explicitly during the optimization/-exploration process (see Figure 10.1). A higher-level algorithm determines an expansion and search strategy to decide which state to expand and which state to select next for further exploration. The chapter starts with the concept of parallel tracking and reconstruction of states by using compressed trace information (see Section 10.1). We continue by presenting our award-winning method for efficiently generating successor states (expanding possible successors for a given state) on GPUs without using the CPU in any way (see Section 10.2).

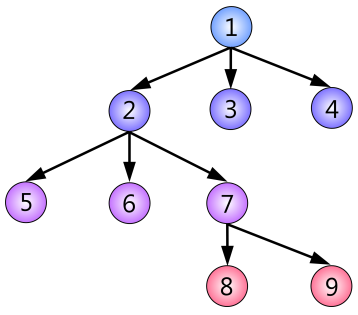

Figure 10.1: Four levels of an exploratory search tree using a form of expansion and exploration strategy [KGK19c].

10.1 Parallel Tracking and Reconstruction of States

Figure 10.2 shows an imaginary and ideal 4-step expansion of a search tree visually separated into its four expansion/exploration steps. In the scope of this figure, different colors indicate different depth levels of nodes. For instance, nodes in purple are the results of an expansion of a root node in blue.

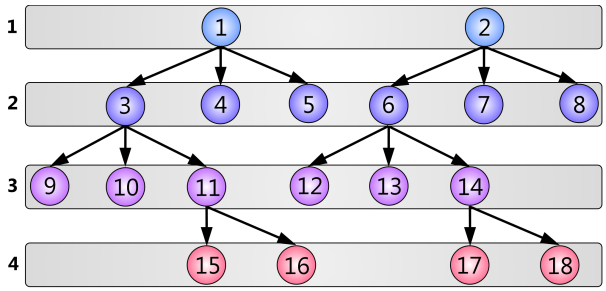

Figure 10.2: Four iterations of an imaginary optimizer using a custom expansion/exploration strategy [KGK19c].

As described in the introduction sections (see Chapter 3 and Chapter 8), the starting point of each arrow refers to a successor-state generation that clones state information and updates variable assignments. The transition to a fully expanded successor state (e.g., from 1 to 3) is achieved afterwards using simulation logic discussed in Part 1 of this thesis. Note that the number of simulation steps performed can also be domain and even state dependent. After performing a certain amount of simulation steps for each state (e.g., a single one for simple board-based games), the state is evaluated, and a heuristic decides whether to explore this state further or to delete it, provided that the target state was not found in this step (see Figure 10.3).

In reality, the actual search trees are usually not expanded in a breadth-first-search manner (see also [CWH08; KGK19c]). This means that the expansion level may not match the depth level of all nodes (the depth of a node in the search tree). As a result, each iteration of the expansion works on nodes that generally do not share much information with each other. A sample for such a case is visualized in Figure 10.4. The heuristic decided that state 11 will be expanded only once instead of twice yielding state 15. After this decision, the domain-specific solver approach also decided to further expand state 4 to a new state 16 in this sample. However, state 16 has search tree depth three, although it was expanded in the fourth step.

Particularly challenging here are scenarios involving backtracking (e.g., discarding state 16 from Figure 10.2 and building a new 16th state from state 4). In such a case, the optimizer "undoes" certain steps by rewinding to an earlier state which may have a higher probability to lead to a potentially better solution compared to the current best known solution. This also means that states must be stored to enable backtracking.

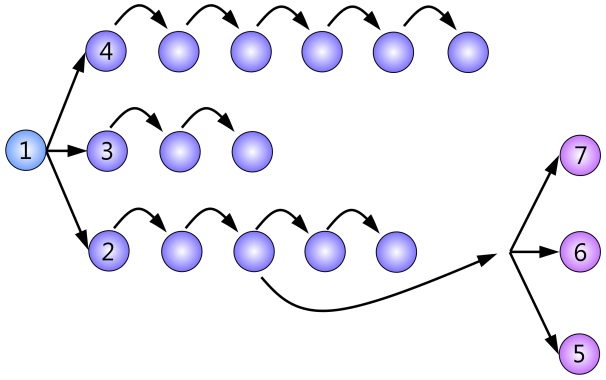

Figure 10.3: Visualized simulation steps in a search tree that has been expanded twice. The simulation logic iteratively changes the originally formed states 2, 3, and 4. After reaching a defined termination condition, the expanded states are evaluated. In this case, the decision was made to unwind several simulation steps of state 2.

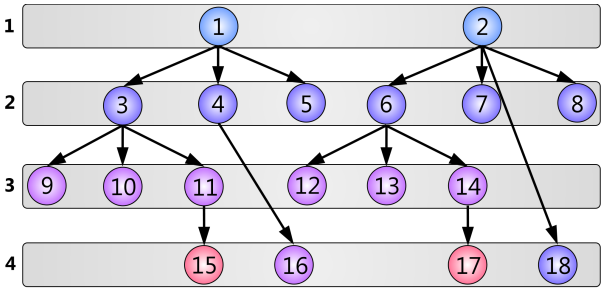

Figure 10.4: Four iterations of an imaginary optimizer using an irregular expansion strategy also potentially involving backtracking [KGK19a].

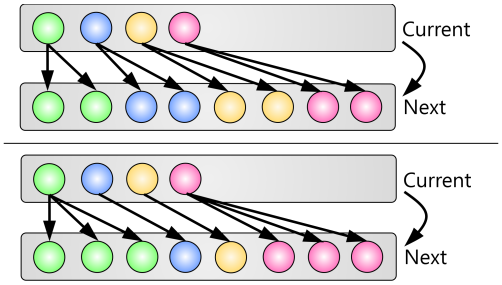

Figure 10.5: Multiple ways of expanding states into the next buffer [KGK19c]. A simple and straight-forward expansion strategy for each state resulting in two successors per state (top) and an irregular state-dependent expansion strategy (bottom). After emitting the upcoming successor states into the *Next* buffer, both buffers are swapped and the *Current* buffer is considered empty again.

In order to make efficient use of available memory and processing resources on the GPU, we conceptually rely on double buffering in our approach and distinguish between a *Current* and a *Next* buffer. For practical reasons, we therefore work with 1D buffers residing in GPU memory. Figure 10.5 visualizes two sample expansion strategies consuming input states from the *Current* buffer and writing their emitted successor states into the *Next* buffer. Figure 10.6 presents a detailed view on an expansion phase followed by multiple simulation iterations.

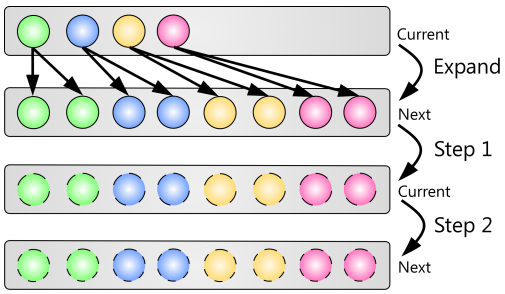

Figure 10.6: Detailed view of multiple iteration steps that are applied to expanded states [KGK19c]. Note that both buffers are continuously swapped between different iterations. This ensures that no information is overwritten and simplifies parallelization. Furthermore, this allows to seamlessly integrate the methods presented in Section 6.2 and Section 6.3.

We present an overview in Figure 10.7 to provide a better understanding of the high-level method we introduced. First, we perform all the necessary memory allocation steps on the CPU and GPU, including page-locked memory transfer buffers. Next, we copy all initial states into the *Current* buffer on the GPU device. Steps 2, 3, and 4 realize the actual optimization loop starting with an initial evaluation step, which in turn begins with an expansion step, as described above. The expansion strategy in this scope is domain and problem specific. However, our method is not constrained by that, since it can work with arbitrarily given heuristics. After performing several evaluation steps to determine the most beneficial states, step 3 involves pruning and history generation. Here, the history-generation and leveraging concept (improving memory consumption) is our key contribution in combination with the related and fully integrated processing pipeline together with improved state-evaluation concepts (e.g., improving occupancy). The idea of our history buffers is to store recovery information in a highly compressed format to avoid maintaining each state in memory. The final step is the reconstruction (or recovery) of states that can be explored next, using the previously identified history information. At this stage, there are two possibilities to continue processing: Either the intended *best* state has been found or we continue processing.

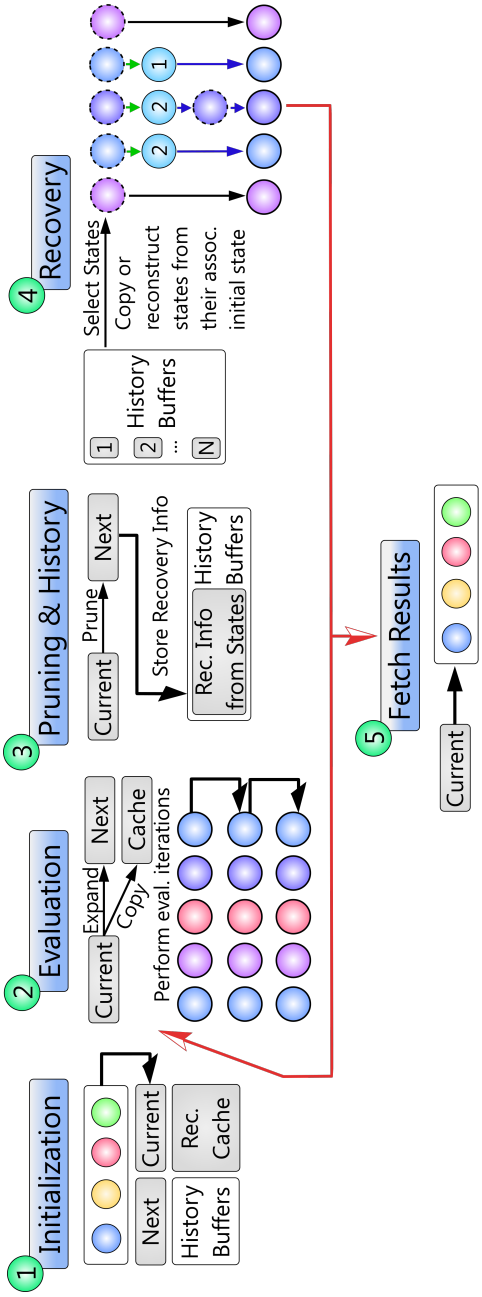

Figure 10.7: High-level workflow of our approach to realize parallel tracking and on-the-fly state reconstruction on GPUs involving 5 general stages [KGK19c]. Our method relies on the concept of double-buffering involving *Current* and *Next* buffers. In addition, we use a method-specific history buffer and a recovery cache.

10.1.1 History and Fill Rate

As outlined above, one of the key components of our method is the maintenance of a state history in an efficient way to reduce memory consumption. For this reason, we introduced the concept of a *History buffer* along with a separate *Recovery* phase to reconstruct states from information stored in the history (see Figure 10.8). Practically speaking, it consists of a single 32-bit and two 16-bit integers per state (sums up to 64 bits) and resides in CPU host memory to be managed by the operating system in the scope of OS-managed virtual memory. The main motivation for using CPU-hosted buffers is the significantly reduced memory pressure on GPUs for large-scale problems. The 32-bit value stores the actual evaluation results (the *rating*) of each state to determine how good the stored state actually is. This information is required to realize a state-selection step based on history information in order to generate new successors from the most promising states (see step 4, Figure 10.7). The remaining two 16-bit integers are used to store the original state index (the source index of the parent state) the current state was generated from and a relative successor index.

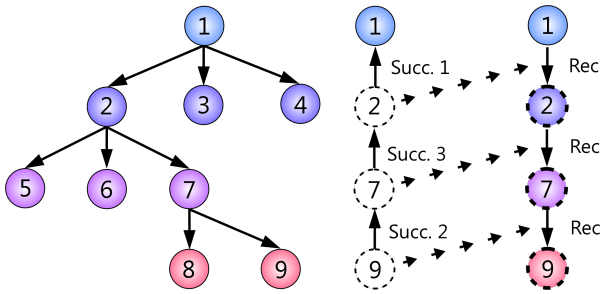

Figure 10.8: Recovery information stored for a certain trace in a search tree (left) consisting of four states (right) [KGK19c]. The trace on the right represents the path from state 1 to 9 including information about the nth successor index per state. The successor information can then be used to reconstruct states from a predecessor state. In this sample, the relative successor indices are 1, 3, and 2. However, the initial source state is required to start the reconstruction process.

During recovery, we can replay a certain path of performed actions using the original state indices and information about the relative successor indices. However, this assumes an efficient method for generating successors based on successor indices. This in turn is a hard problem when it comes to high-performance throughput and efficiency on GPUs. We have addressed this problem in a separate paper, which is described in section Section 10.2 [KGK19a]. The recovery process itself requires a specific source state which which the process begins. From there on, we replay all the decisions made to reconstruct the same state that was previously discarded to reduce consumed memory. It

is worth mentioning that the recovery phase does not cause a large overhead during runtime. This is due to the fact that the number of reconstructed states is usually considerably less than the maximum number of states used during the exploratory expansion phase. Therefore, it is more expensive to keep all the states in memory and move them around than to compress all states and restore a small set of them that survived the pruning step.

Precise information about memory consumption being saved when using our method is discussed and analyzed in the following paragraphs. First, it is important to understand how much memory we will need if we store all states that potentially need to be considered during an optimization run. We denote the maximum number of parallel states during optimization as $|S_m|$ and the maximum number of required states for backtracking in iteration i as $|S_b(i)$. The total amount of memory required for other approaches $M_{other}(i)$ in iteration i is given by [KGK19c]:

$$M_{other}(i) = 2 \cdot |S_m| \cdot m + \sum_{j=1}^{i-1} |S_b(j)| \cdot m, \tag{10.1}$$

where m is memory size in bytes of a single state. In the case of $M_{other}(i)$, we need memory to perform double-buffering and a series of memory allocations to realize backtracking across multiple iterations. This particularly involves storing all backtracking states for all previous iterations due to the fact that we could backtrack to any state from the past in general. However, we have to assume that $|S_b(i)| \approx |S_m|$ to jump to an arbitrary state. In general, this results in all expanded states residing in memory, as in common Monte Carlo Tree Search (*MCTS*) applications [CWH08; Gel+12]. Specifically for MCTS, there are specific algorithmic extensions (such as the work done by Xiao et al. [XMM18]) to reduce memory requirements. This must still be considered a special case of a search-based method. Without loss of generality, this does not hold for an arbitrary method and we still must assume the worst case of $|S_b(i)| \approx |S_m|$.

In contrast to these worst-case memory requirements of other methods, our approach lowers this bar considerably [KGK19c]:

$$M_{we}(i) = 3 \cdot |S_m| \cdot m + |S_m| \cdot i \cdot m_h, \tag{10.2}$$

where m_h is the size of a single history entry; 8 bytes in our evaluation implementation. Since we can safely assume that m_h is substantially smaller than m in practice, we can conclude that $M_{we} < M_{other}$ for a given number of iterations i. However, we optionally add an additional recovery cache that keeps full copies of all states from a specified previous iteration. It covers the most common case where backtracking starts more often from a certain depth level in the tree. Therefore, we require three buffers if the cache is enabled instead of two in comparison to other methods (see Section 10.1.3).

A particular challenge using state-reconstruction is the additional overhead caused by the recovery phase itself. Since all states to be restored must are subject to a repetitive reconstruction that increases in terms of its reconstruction depth, the computational effort increases with the number of iterations. As discussed in the previous paragraph, the number of states to be reconstructed can be considered much smaller than amount evaluated/explored states in parallel. This should already limit the computational overhead, which can be decreased further by using the recovery cache mentioned above.

Nevertheless, maintaining a high occupancy across all states to be recovered is a difficult problem. Mainly because there is repeated switching between the evaluation and variable-assignment steps throughout the recovery phase. Note that this behavior also occurs during state exploration (see Section 10.1 and Figure 10.3). However, depending on the number of look-ahead steps, this may be negligible in comparison to the recovery phase and the current tree depth. Applying kernels to all states that are currently being actively processed leaves certain units unused in all cases due to logically divergent control flow. We have already discussed countermeasures in Section 6.2 using a specially designed algorithm based on thread compaction. However, the same behavior can be observed at higher levels when continuing state-level evaluations or attempting to assign variables in states that are already fully assigned and/or do not require further assignments.

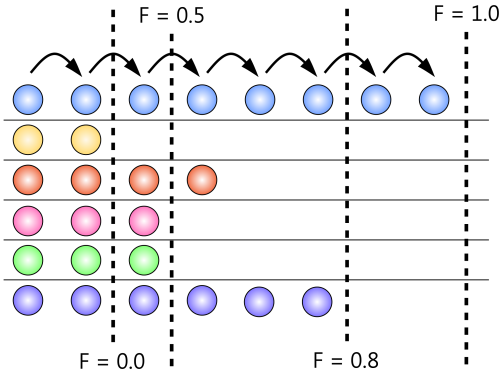

Figure 10.9: Visualization of the fill rate in the presence of 6 traces with a maximum of 7 steps (from left to right) [KGK19c]. Note that this visualization affects both the recovery and evaluation steps. The dotted lines indicate steps after which an assignment step happens. This interrupts the evaluation process to take a different execution path that generates assignments. The longer the recovery phase or look-ahead phase during evaluation, the higher the probability of arriving at the state shown here: Some states will need significantly more steps to be performed than others (similar to thread divergence in GPU programming). The $F \in [0, \ldots, 1]$ values refer to an imaginary fill rate in percent to determine how many states finished processing.

It can be safely assumed that an execution/simulation step is significantly less expensive than an assignment step [KGK19a; KGK19c]. This is due to the fact that (multiple) heuristics will be used in this scope to assign variables according to the current characteristics of the state being processed. The probability of interrupting the simulation step to make variable assignments for all states increases as we have more states in general (see Section 10.1.2 for more information on optimization logic). Even worse, the higher the probability that a single outlier will block further processing, the more states there will be that skip either or both simulation and assignment steps, resulting in unused resources. Consider Figure 10.9, which visualizes this circumstance in the presence of multiple states living in an imaginary buffer. It describes the high level concept of the *fill rate* $F \in [0, \ldots, 1]$ in percent. The larger the fill rate, the longer we will have to wait until states will be scheduled to do assignment steps. Delaying this phase adds additional overhead during simulation, which in turn can be reduced using the presented methods in Part 1. F enables us to have explicit control when an assignment step should happen by taking the number of states waiting for assignments into account. This significantly improves performance throughout our evaluation scenarios (see Section 10.1.3 for more information).

10.1.2 Algorithms

The actual iteration logic to realize an occupancy-oriented iteration loop is shown in Algorithm 17. The inputs to the algorithm are both processing buffers, the current B_C and the next buffer B_N, to realize double buffering. Furthermore, the algorithm also needs access to the active accelerator stream to asynchronously enqueue all operations into the GPU driver worker queue, as well as the maximum number of simulation steps. The first task is to ensure valid variable assignments by performing an initial assignment step in line 1. Afterwards, we enter the main iteration loop that consists of a multiple *reset*, *simulation*, and *assignment* steps. In order to realize a fill rate F implementation, we use a bit set to track per-state activity. Note that this bit set can be efficiently realized using shared memory and a small staging buffer in global memory (with a page-locked counterpart on the CPU side) for global aggregation and data transfer purposes. In the case where this set is not materialized in memory, the *reset* step in line 4 is no longer required. If materialized, this step must reset the bit set so that it semantically contains no elements. The innermost loop in lines 5–10 is required to implement the actual fill-rate logic: As long as the fill rate is not reached, we continue simulating to minimize expensive assignment steps. We then perform another variable assignment step in order to make sure that all states are in a valid and well-defined state (line 11). These steps are reiterated until we reach the user-defined maximum number of steps (number of recovery steps or look-ahead steps in case of evaluation).

Algorithm 17: Our logic for a single optimizer step [KGK19c]

```
   Input: buffer B_C and B_N, GPU stream, max#Steps
1  AssignVariables(stream, B_C);
2  iter := 0;
3  do
4  |  ResetProcessingSet(stream);
5  |  do
6  |  |  iter := iter + 1;
   |  |  /* Note that the simulation logic performs
   |  |     double-buffering and will swap buffers
   |  |     accordingly such that the current buffer B_C will
   |  |     also contain the final simulation results      */
7  |  |  #finished := Simulate(stream, B_C, B_N);
8  |  |  if #finished / |B_C| ≥ F then
9  |  |  |  break;
10 |  while iter < max#Steps;
11 |  AssignVariables(stream, B_C);
12 while iter < max#Steps;
```

Algorithm 18 shows the high level optimization loop that leverages the previously shown Algorithm 17 to implement evaluation and recovery steps. In this scope, the number of evaluation steps is usually greater than the number of recovery steps to implement a buttoned-down way thats avoids easily running into local minima. The inputs to this algorithm are the initial states (often just one) stored in a buffer on the CPU side, referred to as C_I. In addition, the algorithm receives the current GPU accelerator stream, the number of optimization iterations (the maximum tree depth), the maximum number of parallel states, the (bounded) history size, and the number of steps for evaluation and recovery. To begin with, all buffers on the GPU and the CPU side are allocated (lines 1–3) before entering the actual optimization loop (lines 4–20). Within the loop, we first store all recovery states and perform an evaluation step to get the actual state ratings (lines 5–7). The next step is pruning of all

Algorithm 18: Our algorithm for state tracking and reconstruction on GPUs [KGK19c]

Input: CPU input buffer C_I, GPU stream, #iterations, max#States, max#HistoryEntries, #evalSteps, #recoverySteps
Output: output states on CPU

```
1  B_C, B_N, R := Allocate<GPU>(max#States);
2  B_H := Allocate<CPU>(max#HistoryEntries);
3  CopyTo<GPU>(stream, C_I, B_C);
4  for iter := 1 to #iterations do
       /* Prepare fast recovery and evaluate states          */
5      Copy(stream, B_C, R);
6      Evaluate(stream, B_C, B_N, #evalSteps);
7      Swap(&B_C, &B_N);
       /* Prune the current states                           */
8      Prune(stream, B_C, B_N);
9      Swap(&B_C, &B_N);
10     Synchronize(stream);
11     AddToHistory(B_H, B_C);
12     recoveryIndices := ResolveRecovery(B_H);
       /* Recover states from the beginning                  */
13     recovery := recoveryIndices \ GetStateIndices(R);
14     ScatteredCopy(stream, C_I, B_C, recovery);
15     if |recovery| ≠ 0 then
16         Recover(stream, B_C, #recoverySteps * (iter − 1));
17     end
       /* Recover states from fast recovery cache           */
18     ScatteredCopy(stream, R, B_C, recoveryIndices ∩ indices(R));
       /* Perform recovery of current iteration             */
19     Recover(stream, B_C, #recoverySteps);
20 end
21 result := CopyTo<CPU>(stream, B_C);
22 Synchronize(stream);
```

states based on their determined rating, after which we insert the surviving states into the history buffers on the CPU side (lines 8–11). Depending on the decision of which states to recover, the states must either be reconstructed (lines 12–17) or are fetched from the fast cache (line 18). In the first case, we must perform a number of `iter-1` recovery steps to ensure that they represent a comparable state in relation to those stored in the cache. Before proceeding to the next optimizer iteration, we perform the specified number of recovery steps on all states. This again makes sure that all states, whether fully reconstructed or copied from the cache, are prepared for the next iteration. The last step of the algorithm is to issue a copy command to fetch all remaining "best" states from the GPU device.

Note that this algorithm requires two synchronization points between CPU and GPU (lines 10 and 22). In these cases, we need to retrieve information from the GPU and transfer it to the CPU. This is not required in the case of a CPU to GPU transfer of CPU buffer contents C_I in line 14, since this buffer is considered to reside in read-only page-locked memory.

Implementation Details

The following section summarizes implementation details that are important when reimplementing our method. These insights described here help to reproduce the benchmarks and measurements shown in Section 10.1.3.

We used the ILGPU [Kös23] compiler in combination with C# to develop our prototype and all GPU kernels. Loading history entries back into CPU memory leverages page-locked allocations to enable direct-memory accesses from the GPU side to improve performance. State data is represented via SOA-based (structure-of-array) memory allocations to improve memory-transfer efficiency on the GPU in all kernels. In addition, the `ScatteredCopy` function is realized via explicitly specialized kernels that efficiently perform scattered copies using reordering masks. Note that the pruning step happens on the CPU in our implementation, as this allows using conveniently modeled high-level state comparisons without considering GPU-programming constraints.

10.1.3 Performance and Memory Consumption Evaluation

We followed a similar evaluation approach as in previous work, using artificially generated workloads to avoid difficult-to-understand and non-reproducible benchmarks [KGK19c]. As current state-of-the-art optimization systems[1] leveraged the power of neural networks to guide optimizers by rating assignment possibilities, we followed this high-level concept. Consequently, we made use of matrix-matrix multiplications to model our assignment steps (see Section 6.2.3 for details of how many operations our matrix-matrix multiplications involved) [KGK19c]. Due to the fact that these multiplications allowed for side-effect-free realization of computational workloads, we also implemented the interpreter steps by reusing the same concept.

Typically, the workload of an assignment step is significantly larger than that of an interpreter step [KGK19c]. A well known example of this setting is the evaluation of a large-scale neural network to perform assignments [Gel+12]. To be able to represent and emulate such scenarios, we introduced two load factors l_1 and l_2 that control the interpreter and assignment workloads, respectively. In order to follow known optimization systems, we always assumed that $l_2 >> l_1$, and thus introduced the following load relation by treating l_2 as the additional overhead that an assignment step has over an interpreter step. Hence, the actual computational *load factor* of an assignment step is referred to as Load $= l_1 \cdot l_2$. We further set the assignment *load factor* l to be at least 10 to achieve a reasonable workload per step [KGK19c].

There were some additional peculiarities of optimization systems from practice to consider: While they tracking multiple states in parallel, they did not always break the interpreter loop at the same point in time to ask for assignments (see also Figure 10.9). Without modeling these details in the context of our benchmarks, all states would either always break for assignments at every step or once every few steps. Since this behavior is completely unintended, we introduced varying probabilities for each state. These probabilities allowed us to implement common domain-dependent characteristics that required an assignment from time to time (referred to as P). The larger P was, the more likely that state would need an assignment step during a run of our evaluation scenarios.

We evaluated our concept of parallel tracking and reconstruction using two GPUs from NVIDIA featuring different compute capabilities [NVI23a]: GeForce GTX Titan X and GeForce GTX 1080 Ti [KGK19c][2]. Here, each performance number was determined by taking the median execution time of 100 runs to compensate for overhead from the surrounding benchmark environment. The fill rate F was set to values $\in [0, 30, 60, 100]$, where a value

[1] At the time of writing this thesis.

[2] Note that both GPUs were state-of-the-art at the time of the publication. Therefore, we retained these measurements to present the original results. However, it is of interest to reevaluate our method on more recent GPUs in future work (see Chapter 14).

of $F = 0$ means that our model immediately pauses for assignments when a state requested a variable assignment. This automatically served as a baseline against which we could compare our improvements: Increasing F should eventually lead to significant performance improvements. In this context, a value of $F = 100$ meant that we were lazily waiting until we reached a point in time where all states were waiting for allocations.

The actual evaluation of our approach is divided into two parts: The first part focuses on runtime improvements using our state-tracking (fill-rate based optimizer loop) and the second part compares memory consumption and runtime overhead using our state reconstruction method.

State Tracking Performance

In this part, we considered evaluating parallel tracking runtime performance within an optimizer iteration. This was sufficient and representative, since all necessary simulation and assignment steps take place within a higher-level optimizer step (see also Algorithm 17). In general, increasing the number of optimization iterations lead to a linear increase in runtime, since all iterations of the optimizer were performed sequentially one after the other. Consequently, we focused on parts of the optimizer covered by our proposed method.

A single optimization step was configured to use a Runge-Kutta scheme to simulate multiple steps to explore the search space, determine the path to track, and ultimately "move forward" with a smaller number of steps than the steps previously used for exploration. We did 8 evaluation steps for exploration in each step and one step to "more forward" (our recovery step) [KGK19c]. Figure 10.10 shows speedups achieved with our tracking approach while the *load* factor was set to 10 while using 16384 and 65536 states in parallel [KGK19c]. These state configurations were chosen to realize reasonable workloads on the devices. Using a smaller number of states usually makes GPU acceleration unnecessary because of the overhead of maintaining GPU buffers and implementing and maintaining GPU-aware implementations in general. The speedup compared to an optimization loop without fill-rate optimizations ranged from 2.4× to 2.7× on both GPUs, while a significant 4× increase in the number of states did not yield further speedups when F was set to 30%.

By setting F to 60%, we were able to achieve speedups in the range of 2.95× to 3.7× on both GPUs, depending on the benchmark configuration. This picture did not change significantly when F was set to 100%. This was due to the fact that the majority of the 16384 states were already pausing in each step even when $P = 30\%$ (the worst case in general), since they were waiting for assignments. Therefore, setting P to 70% did not result in any further noticeable improvements for our method. Interesting to mention is the very similar scaling behavior on both GPUs although they differed in many properties. This could give the false impression that there were no performance differences between both GPUs.

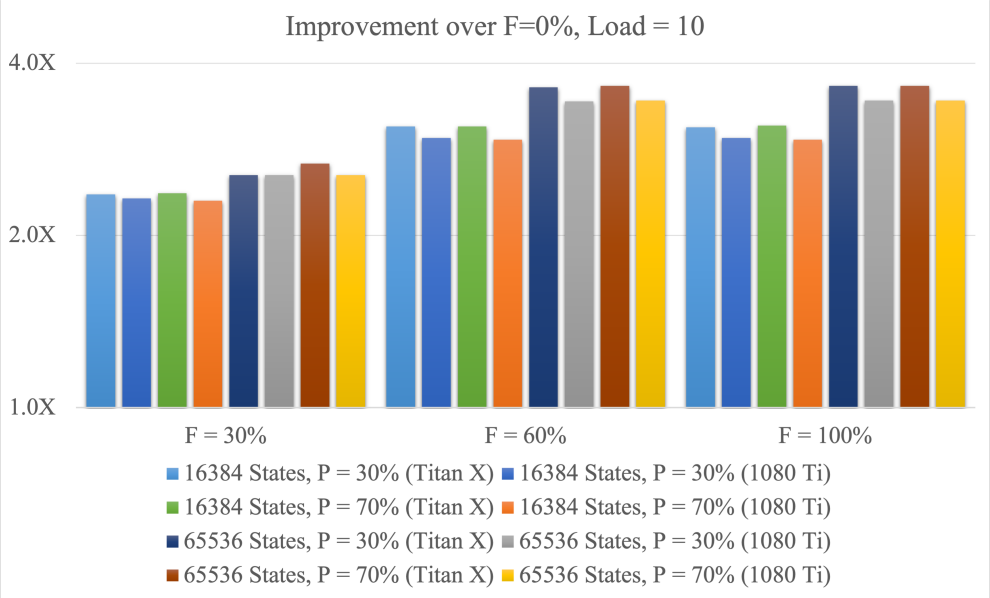

Figure 10.10: Speedups for different assignment probabilities P, number of states, and fill rates F on both evaluation GPUs (log2 scaling, higher is better, based on performance numbers published in [KGK19c]). Note that the computational load was set to 10× for all measurements. The speedups were calculated by comparing the gathered performance measurements to breaking the interpreter loop for assignments after each step ($F = 0\%$).

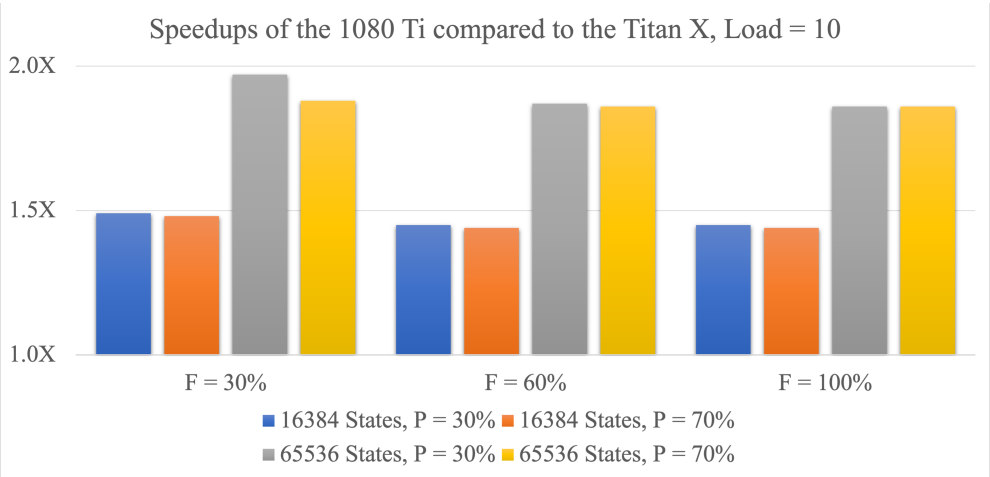

Figure 10.11: Speedups achieved by using the GTX 1080 Ti compared to performance numbers measured on the GTX Titan X in the presence of different number of states and value P (higher is better, based on performance numbers published in [KGK19c]).

In fact, there were significant runtime differences between the two evaluation GPUs. Figure 10.11 shows speedups measured using the GTX 1080 Ti compared to the older GTX Titan X. The newer GTX 1080 Ti outperformed its predecessor generation GPU by at least 1.45× using 16384 states. Increasing the workload by using 65536 states resulted in a huge speed increase, ranging from 1.85× to 1.97×. When looking at the overall acceleration measured on both GPUs, it was found that varying the computational load provided further speed increases of 2× (max) compared to a non-optimized step implementation.

Figure 10.12 presents the effect of varying the computational loads while fixing our assignment probability to $P = 70\%$ using 16384 and 65536 states in parallel. Overall, we measured speedups of 3.3× up to 3.7× using $F = 30\times$ and 5.2× to 7.2× using $F \geq 60\%$ on both GPUs. However, the overall relative scaling behavior remained the same when varying the computational load. Note that this was not true for the overall runtime which behaved highly similarly to the comparison shown in Figure 10.11 [KGK19c].

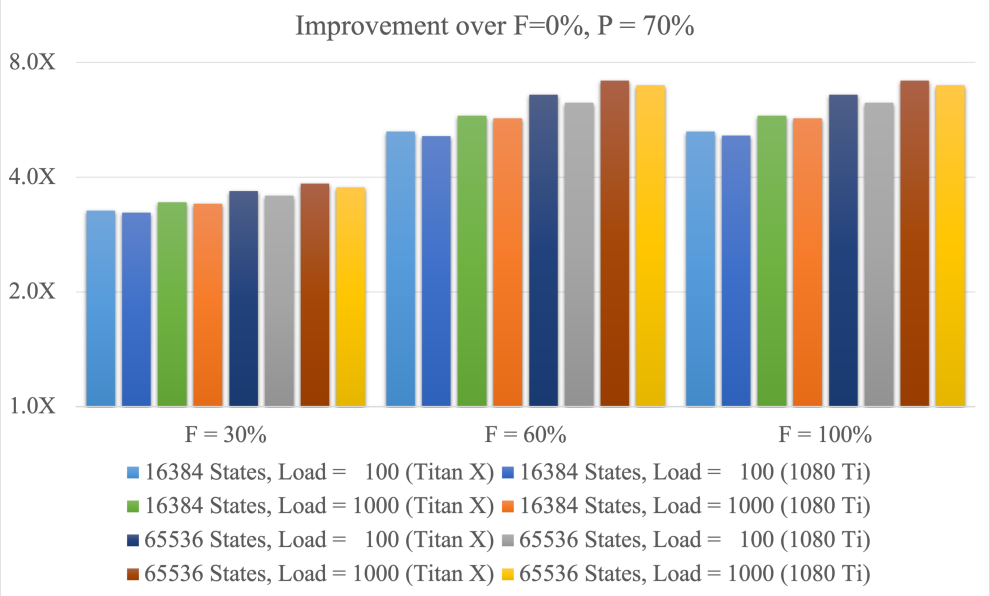

Figure 10.12: Speedups for different computational loads (100× and 1000×) and fill rates F while setting $P = 70\%$ on both evaluation GPUs (log2 scaling, higher is better, based on performance numbers published in [KGK19c]). The speedups were calculated by comparing the gathered performance measurements to the breaking the interpreter loop for assignments after each step ($F = 0\%$).

State Reconstruction Performance

The second part of the evaluation starts with changing the benchmark setting and considering a situation where the optimizer is in iteration 16. We further consider the case where we had to backtrack to iteration 15. Moreover, we assumed that we must remember all visited states which is a common use case (see also Section 10.1.1) [KGK19c].

As described in Section 10.1.1, a single history entry required 8 bytes to store all recovery information necessary to reconstruct a state. In addition, we had to keep all states in memory that were requires for parallel processing. This means that we had to maintain at least two buffers storing all state information (for double-buffering purposes) and another copy when using the fast recovery cache. Figure 10.13 shows the total memory consumption in GB of our method with and without caching compared to a method maintaining all states in memory for backtracking purposes. As indicated in the diagram, our method consumed several orders of magnitude less memory than traditional methods. This was already a huge improvement when only 16384 states were used in parallel.

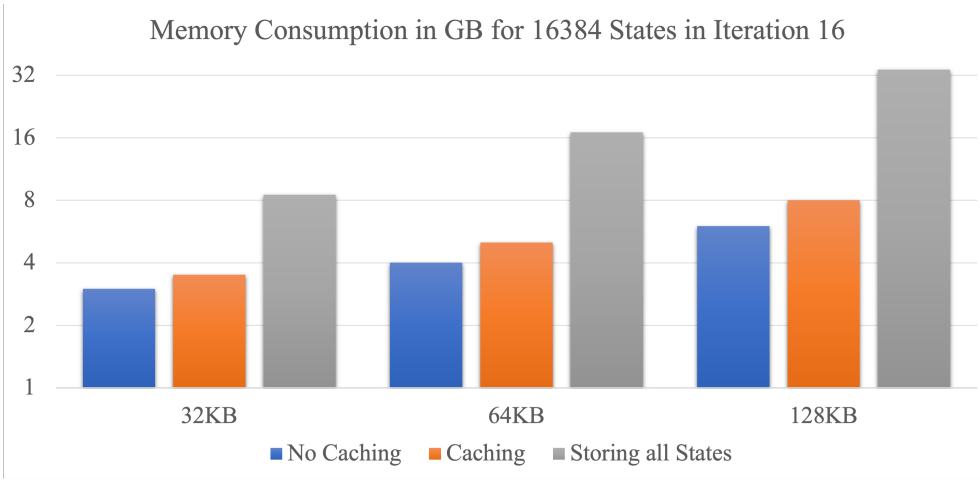

Figure 10.13: Total memory consumption in GB for different state sizes (32KB, 64KB, and 128KB per state) while using different tracking approaches in iteration 16 (log2 scaling, lower is better, based on performance numbers published in [KGK19c]). This figure compares storing all states in memory to using our method with and without iteration caching enabled.

Unfortunately, on-the-fly state reconstruction is not free, as it can require many replay steps that block our processing pipelines before we proceed to the next step. Figure 10.14 presents speedups and slowdowns of our method in comparison to keeping all backtracking states either in CPU or directly in GPU memory. Our method was approximately 3.5× slower than storing all states in

memory when caching is disabled and the state size is set to 32KB. Increasing the state size immediately exceeded available GPU memory on our GTX 1080 Ti and caused us to fall back to a CPU-based buffer. Comparing our reconstruction performance to the CPU-based buffers yielded huge speedups of 6.2× to 7.2× [KGK19c]. We were able to achieve these improvements because our reconstruction buffers fit easily into GPU memory. It is also worth mentioning that the number of states can be increased by several orders of magnitude until we hit memory capacity limits on modern GPUs [NVI23a; KGK19c].

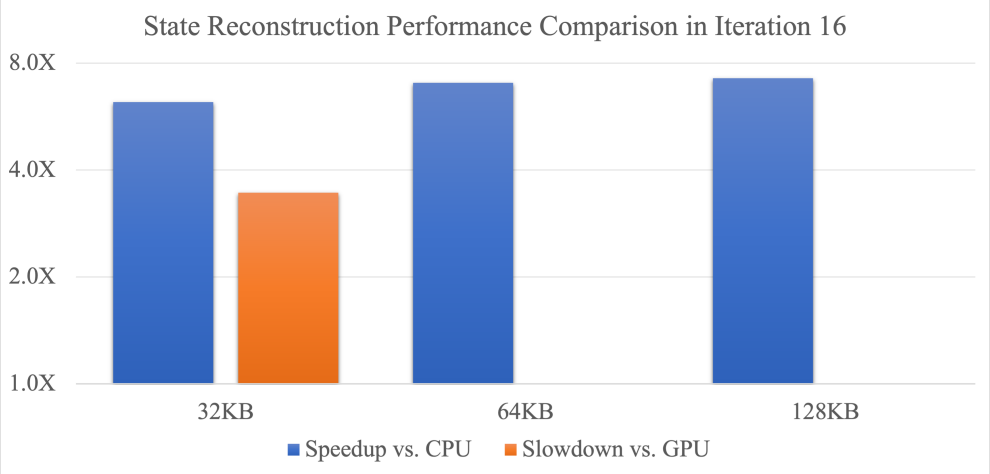

Figure 10.14: Performance comparison of our state reconstruction to storing all states in CPU memory and copying them over to the GPU for processing (speedups, higher is better, highlighted in blue) and keeping all states in GPU memory in iteration 16 (slowdowns, lower is better, highlighted in orange). The diagram also shows comparisons between different state sizes (32KB, 64KB, and 128KB per state) to be tracked (log2 scaling, based on performance numbers published in [KGK19c]). Note that we used the GTX 1080 Ti for this evaluation which is limited to 11GB of vram [KGK19c; NVI23a].

10.2 Fast and Efficient Successor State Generation

As discussed earlier, successor-state generation is an essential task in the context of heuristically driven optimization systems (see also Section 1.2 and Chapter 10). Such a function for successor generation or neighborhood exploration is denoted $N(x)$, where x represents an optimization state for which successors are to be generated [KGK19a]. The conceptual idea of using such a function is illustrated in Figure 10.15.

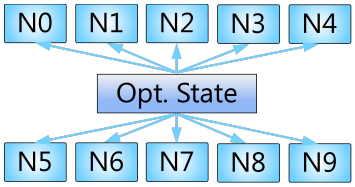

Figure 10.15: Visualization of $N(x)$ that conceptually generates ten potential successors or *neighbors* for a particular *optimization state*.

Formally, a state x is composed of state and domain dependent variables λ_i, where $i \in \{1, \ldots, |\lambda|\}$ and λ is the set of all variables that can be assigned in a single state. We assume that all states have the same number of variables during the solution of the optimization problem. Without loss of generality, we define the solution of an optimization problem we wish to solve as finding an optimal assignment of all variables λ_i to values V_j, where $j \in \{1, \ldots, |V|\}$ and V is the set of all possible values that can be assigned to a variable. We further assume that all variables have the same set of values. Consider the following example, where we encounter two variables with different value sets in their original problem definition to assign these variables to

- λ_1, which can only be assigned to $\{1, 2\}$, and
- λ_2, which can only be assigned to $\{2, 3, 4\}$.

A simple, straightforward and theoretically sound solution would be to set $V = \{1, 2, 3, 4\}$ in order to represent all values of the initial problem definition. However, this would result in a large number of non-possible assignments of variables to values that must to be discarded in practice. Consequently, we assume that all values V_j represent an index that in turn points to their jth assignment possibility. This assumes an input mapping Y [KGK19a],

$$Y(\lambda_i, V_j) \mapsto j\text{th value which can be assigned to } \lambda_i,$$

which in turn causes V_j to become an alias for index j ($V_j \equiv j$). In this sample, this results in $V = \{1, 2, 3\}$ and a mapping Y defined as

$$Y(\lambda_1, 1) := 1, Y(\lambda_1, 2) := 2, Y(\lambda_1, 3) := \perp,$$
$$Y(\lambda_2, 1) := 2, Y(\lambda_2, 2) := 3, Y(\lambda_2, 3) := 4,$$

where $\perp$ refers to an illegal assignment. Further, the assignment of $\lambda_1 \mapsto 3$

would be considered invalid, as there is no such assignment possibility since $Y(\lambda_1, 3) = \bot$. Hence, to reject invalid assignments by definition (without paying attention to additional constraints), a simple boundary check is sufficient in all cases. We refer to the number of valid assignment possibilities for a given variable λ_i by $|Y(\lambda_i)|$ [KGK19a]. Note that depending on domain/scenario specific constraints, not all of these variables may be *free to assign* at all times.

The goal is to solve large-scale problems consisting of thousands of states, which in turn are composed of thousands of variables in total. Therefore, it is not feasible to enumerate all states in the immediate neighborhood of a given state x and materialize them in memory. After several expansion steps, the number of assignment possibilities grows exponentially, making such a solution infeasible in practice. Instead, the idea is to explore *more promising* states in favor of all possible successors. This refers to the fact that the probability of such state leading to the optimal solution is considered to be higher than the probability of other possible successors. We follow related work from Munawar et al. [Mun+09] and Campeotto et al. [Cam+14b] to explore the neighborhood of each variable locally. Unlike other work, we do not randomly choose variables and their possible assignments, but rely on a local rating concept for possible variable assignments. We call this "guided by *local heuristics*" to narrow the exploration space while guiding the neighborhood investigation process in a promising direction. Local means that each variable assignment rating is performed independently per variable without taking other assignments to other values in the same state at the same time into account. This works by interpreting local rating evaluations for each assignment possibility per variable as a weighted importance-sampling problem (see also Figure 10.17). This requires summing up all local ratings per variable given by a heuristic-based evaluation function H [KGK19a]:

$$R^S_{\lambda_i}(x) := \sum_{V_j \in V} H(x, \lambda_i \mapsto V_j), \tag{10.3}$$

where $R^S_{\lambda_i}(x)$ represents the sum of all local assignment ratings of the variable λ_i in the scope of state x. To select a value assignment, we first choose a uniformly distributed random value $r_k \in [0, \ldots, R^S_{\lambda_i}]$. We then remodel the presented sum as an inclusive prefix-sum to retrieve offsets based on the relative importance of all local ratings in relation to all others [KGK19a]:

$$O^S_{\lambda_i}(x, k) := \sum_{j=1}^{k-1} H(x, \lambda_i \mapsto V_j), \tag{10.4}$$

where k is the index of the current value V_k for which we want to compute the offset [KGK19a]. Finally, we pick a matching value V_k for which

$$O^S_{\lambda_i}(x, k-1) \leq r_i < O^S_{\lambda_i}(x, k) \tag{10.5}$$

holds. This enables us to parallelize the rating process for a variable λ_i by rating all possible mappings to values in V in parallel. Unfortunately, it is

also possible that the assignment of $\lambda_i \mapsto V_j$ and $\lambda_k \mapsto V_l$, where $i \neq k$, leads to a violation of given constraints. To avoid such cases in general, we *do not* assign different variables in parallel. On the one hand, this solves the described problem. On the other hand. it also allows using sophisticated heuristics that take all other variable assignments into account when computing local ratings,. Note that this is not a restriction, as the order in which variables are assigned can be either random or based on some other meta-heuristic decision process. This avoids the appearance of biased optimization results and enables the optimization system to recover from local minima in certain situations and domains.

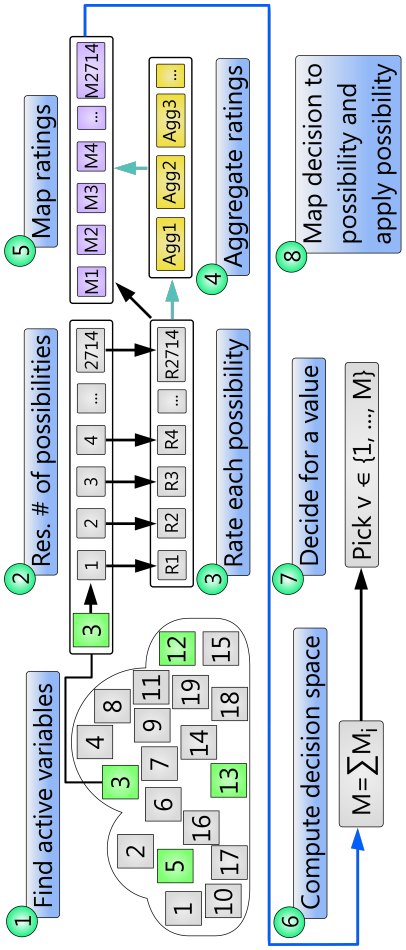

Figure 10.16: High-level view of our approach on a conceptual level involving eight steps [KGK19a].

Figure 10.16 shows our high level workflow in a non-parallel and conceptual way. In the first step, we detect *active variables* that require an assignment. From now on, we differentiate between *active* and *inactive* variables. The latter are assumed to keep their current assignment and do not need to be reassigned. The decision whether a variable is considered *active* is usually made by the parent optimization process. An example would be a chess engine requesting

new assignments for all chess pieces that can be moved. Therefore, we assume that this step has already been completed and the remaining task in this scope is to find these variables.

After identifying all active variables, we determine an order in which we plan to assign them. This can either be done in a random order or based on an even higher-level heuristic. In the diagram shown, variable 3 had been selected first (to be assigned). No matter in which order all active variables are assigned, the next step is always to resolve the number of possibilities. We have already discussed the possibility that the value set may contain too many values, and we use boundary checks to immediately skip values that are trivially not assignable. Depending on the current state, more values may not be assignable. For this reason, we resolve the number of assignment possibilities based on the current state and the optimization model.

We evaluate each possibility with a local heuristic function H as described previously in step 3. This results in a set of rating values R of a user-defined type. For the algorithm itself, the type of rating values is not important and is considered opaque. A common choice in practice are 32-bit or 16-bit float or 16-bit float values [KGK19a].

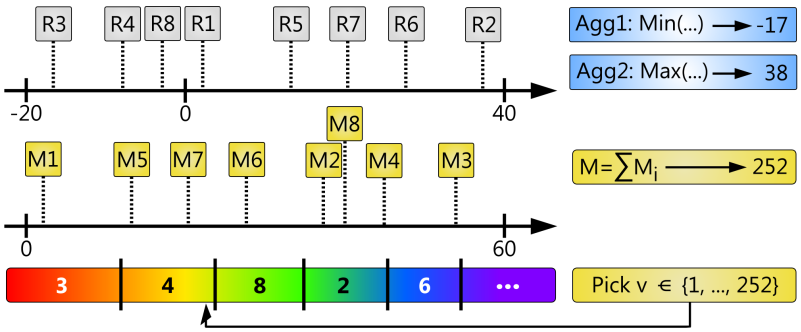

Figure 10.17: This figure demonstrates a sophisticated mapping process. A set of eight determined ratings with values between -20 and 40 based on imaginary user-defined rating specification (top) [KGK19a]. Before converting these ratings to our algorithm-internal mapping space M, we apply two provided aggregators which resolves the minimum (-17, *Agg1*) and maximum value (38, *Agg2*). In this sample, positive ratings are considered less important than their negative counterparts. Therefore, the mapping process uses the minium and maximum information to mirror the negative values from the rating level to the right-hand value range of all M values (middle). The remaining two steps are the sum computation and the selection process of the random decision variable (bottom). Color coding of different intervals refers to the relative importance of all mapped ratings, with red being the most important one.

In order to reason about the individual ratings R, we convert all ratings to values $M \in \mathbb{N}$ (step 5). If a mapped rating M is 0, this possibility is considered impossible or not allowed and therefore ignored during the assignment process. The mapping process itself allows domain experts to model custom activation-function like behavior for variables, taking aggregated information into account (step 4). All remaining steps (6 to 8) use the formally defined importance-sampling procedure. Once all mapped values have been determined, we compute the so called *decision space*, which is the sum of locally determined mapped ratings (formally defined in Equation (10.3)). Based on this sum, we pick a random value and select the corresponding importance interval based on all M values (see also Equation (10.4) and Equation (10.5)). A practical algorithm and implementation-aware visualization is shown in Figure 10.17.

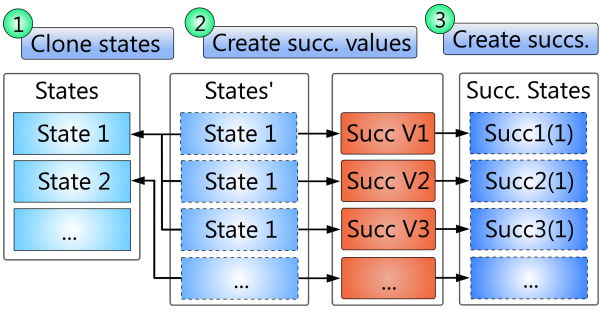

Figure 10.18: Our three step successor generation process [KGK19a].

In analogy to our method of parallel state-tracking (see Section 10.1), we also use double buffering to realize successor generation. It works in three steps (see Figure 10.18), beginning with an initial cloning of all *source* states into the *next* buffer. In practice, a single source state is cloned into the target buffer multiple times [KGK19a]. In this way, users have full control over the state-expansion or growth rate in the context of one iteration. The second step involves the generation of *successor values* for variable-assignment, tracking and recovery (see also Section 10.1.1). A successor value acts as a random seed for variable assignments in general. Storing this value together with the origin-state information allows us to always recover the same state based on the stored history entries. Generating these values can be realized via a parallelized algorithm using an random-number generator to determine different values for all states. The final step in the actual successor creation, which uses our presented high-level algorithm Figure 10.16 and swaps the *current* and *next* buffers.

10.2.1 Detailed View

For efficiency reasons, the successor-generation functionality is designed to be realized as a single GPU kernel that incorporates all local heuristic evaluation functions and mapping aggregators within it. By integrating all necessary functions within the same kernel, we can ensure seamless and efficient execution of our algorithms on modern GPU architectures. As shown in Figure 10.19, our warp-focused (or wavefront-like) workflow provides an in-depth view of every step involved inside our kernel. This visualization helps differentiate between parallel processing steps, with group-wide operations running independently on each thread or being subdivided into warp-sized chunks.

In practice, this can be achieved by meta-programming techniques like template/generic specialization, or partial evaluation [Kös+14a; Kös+14c]. Methodologically, we use a single thread group with N threads per optimization state that assigns all variables, making heavy use of thread-group-wide cross-thread communication. Step 1 involves the use of bit sets that can be realized with densely packed 64-bit long values to improve memory efficiency. Determining all active variables means parallelized bit-set-based aggregations across all threads in the group. Depending on the scenario, it may be more advantageous to use only threads in the first warp. This leaves more free space for other concurrently running groups in the scope of the warp scheduler on the device. Afterwards, we iteratively assign each free variable one after another via group-wide broadcasts (see above). Note that this step is highly dependent on the optimizer implementation, as these systems usually already keep track of variables that need to be assigned.

In step 2, we determine the number of possibilities in parallel for each thread in the group, since this can be considered a cheap pre-variable operation [KGK19a]. Therefore, it is usually not necessary to involve group-wide value broadcasts that force synchronization. If determining the number of possibilities turns out to be more expensive, it can also be parallelized in this step and propagated to all other threads in the group. However, this step is not considered the bottleneck of the algorithm.

Rating (step 3) of each possibility is performed by all threads in parallel using group-stride loops. The *rating storage* holds all computed individual ratings R_i determined using the local heuristics introduced earlier. In general, the storage is set up in global memory if a recomputation of all heuristic values is more expensive than loading values asynchronously from memory. Moreover, using vectorized load/store operations gives an excellent memory throughput due to coalesced memory accessed by the design of the kernel structure. This also allows us to hide memory latency when solving large-scale problems, making the overhead of storing values in global memory at this stage negligible in context of the whole processing pipeline. Step 3 also allows caching of heuristic-related values in shared memory, since we do not use any shared memory at this stage without our approach.

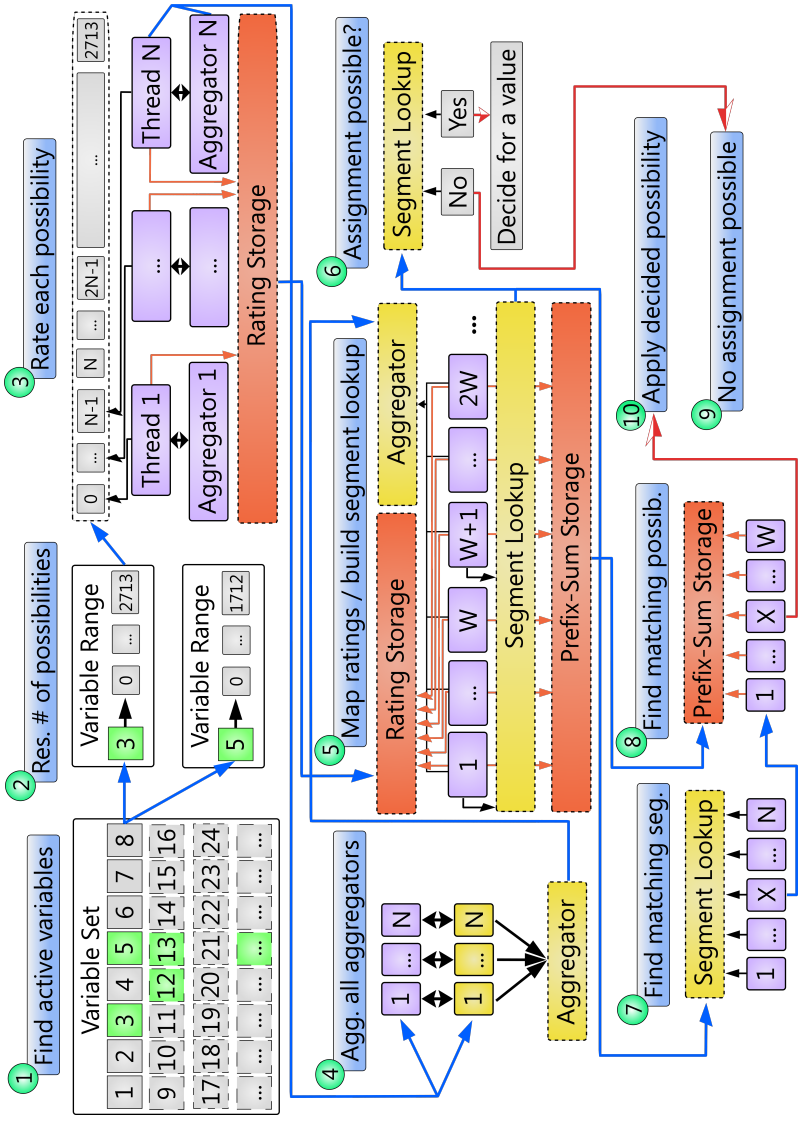

Figure 10.19: Detailed view of our method involving ten different steps and cross-thread interaction via shared memory and warp-wide reductions [KGK19a]. Multiple conceptual phases enable us to make efficient use of all available hardware processing resources. Blue arrows refer to data dependencies and data flow, whereas orange arrows refer to memory accesses. Areas surrounded with dashed lines represent temporary information stores available to each thread in the group. Purple boxes are thread-local contents in register space, yellow boxes represent data stored in shared memory, and red boxes is data stored in global device memory.

Individual aggregator instances (in purple) are kept in register space (step 3) until aggregation (upcoming step 4). To map all stored ratings, we need accumulated aggregation information, which is achieved in step 4. It works by aggregating all intermediates at the warp and then at the group level using broadcasting via shared memory.

The actual mapping step of all ratings is done by splitting the process into chunks of the size of a warp while accessing the previously computed rating and aggregator information. Each chunk processed by a warp (referred to as a *segment*) converts ratings into mappings in an iterative group-stride manner. Note that the way in which values are loaded from the rating storage still happens in a coalesced way to improve memory efficiency. Every warp computes a segment-wide offset table (see also Equation (10.4) and Figure 10.17) over a prefix sum and saves all relative offsets of each element in the global prefix-sum storage, which does not fit into shared memory. At the same time, the first lane of each warp stores the already computed rightmost prefix-sum offset in shared memory. These offsets represent an acceleration structure in shared memory that is then used to determine the selected assignment possibility in the end (steps 6, 7, and 8). This avoids the use of expensive global memory IO for segment offsets while providing fast and efficiency random-access lookups of mapped rating values.

In our case, we use a single 64-bit integer variable per warp (per segment) to store the rightmost upper bound. We recommend using 64 bits per entry to avoid overflows on large-scale problems and problems with larger mapping values that can cause their sums to overflow 32-bit integers [KGK19a]. It also leaves room for further case-by-case differentiations or compile/runtime specializations of our method to increase the number of segments cached. Conservatively, we can assume about 48 to 80kb per group on a current state-of-the-art NVIDIA accelerator [NVI23a] with group sizes of 1024 threads each (2 groups per multiprocessor to achieve maximum thread occupancy). This yields about

$$\frac{48 \cdot 1024}{8} \cdot 32 = 196608 \text{ using a GPU with compute capability 7.0 and}$$
$$\frac{80 \cdot 1024}{8} \cdot 32 = 327680 \text{ using a GPU with compute capability 8.0}$$

supported assignment possibilities per variable, which covers even large-scale problems.

Using the segment lookup, we can easily determine in step 6 whether an assignment is possible or not. If the right-most value in the segment lookup is non-zero, an assignment is possible since there is at least one solution. If this is the case, we proceed with step 7 by choosing a random decision variable r which is smaller than the right-most boundary stored in the segment prefix-sum lookup (see Figure 10.17). If we cannot assign the variable, we continue with step 9 in which the parent optimization system is notified that no assignment was possible. Depending on the domain, this can also mean that no solution

could be found due to conflicting constraints. Should this be not the case, we can deactivate the variable and continue with the next active one.

If the rightmost value in the segment lookup is non-zero, an assignment is possible because there is at least one solution. If this is the case, we proceed to step 7 by choosing a random decision variable r that is less than the rightmost boundary stored in the segment prefix sum lookup (see Figure 10.17). If we cannot match the variable, we proceed to step 9, which informs the higher-level optimization system that a match was not possible. Depending on the domain, this may also mean that no solution could be found due to conflicting constraints. If this is not the case, we can deactivate the variable and continue with the next active variable. At this stage, we can cache the heuristic data in shared memory since it will not become a problem.

The 7th step uses all threads in the group to determine the warp index to which the decision (the random value drawn) belongs. The group performs a stride loop, accessing our previously built shared-memory-based lookup for segments offsets. Due to the nature of shared memory, we benefit greatly from the memory-access times in this step. From there on, we send information of the selected segment index to the first warp in the group, freeing up processing resources at this stage (step 8). All lanes in the first warp then iteratively load the selected subsection of prefix-sum values from global memory. By definition, one of these lanes will sooner or later find the matching decision interval in step 8. At this point, there is only one step left: applying the decision in step 10 and conceptually deactivating the current variable being processed. After that, the processing of the next variable considered active in the queue continues.

It is important to also consider duplicate states that can theoretically arise from this process. There are several reasons for this, one of which is randomized successor generation with generated seeds. The other is the fully parallelized state generation with no communication between states. Hence, it can be argued that this is the major drawback of the presented method. In general, the probability that a duplicate occurs depends on the constraints, the variables (as well as their assignment possibilities), and the definition of the local heuristics. In fact, the probability that two states are identical after n expansion steps is p^n, where $p \in [0, \ldots, 1]$ is the probability for the occurrence of a duplicate state [KGK19a]. Therefore, this fact can be safely neglected for practical large-scale applications [KGK19a].

10.2.2 Variable Types and Memory Consumption

To provide more flexibility to the users, we distinguish between different types of variables called λ^T. This allows them to use different heuristics in a convenient and efficient way by specialized the proposed method accordingly [KGK19a]. Furthermore, this also allows the use of different rating types R, aggregators, and even mapping types M (see Section 10.2.1). This decision could also be made in the scope of a single provided heuristic. However, this would again lead to diverging code paths within this code and increase maintenance overhead due to the strong coupling of separable modules [Kös+14a].

In terms of memory required to use our method, we need intermediate buffers to store all prefix-sum and rating values. This results in a single 64-bit integer value to store prefix-sum offsets, and it also depends on the size of each custom rating entry. As outlined in the section above, we support multiple variable types, which in turn may depend on their rating type. We want to minimize the memory footprint and therefore reuse the intermediate buffers for the valuation. Hence, the size of each rating entry to allocate is the maximum size of all rating types being used [KGK19a]:

$$\text{size(R entry)} := \max\Big(\text{size}(R_1), \ldots, \text{size}(R_{|\lambda^T|})\Big), \tag{10.6}$$

where $|\lambda^T|$ is the number of different heuristics λ^T being used. Since we require a single rating entry per variable assignment possibility, the overall memory consumption in bytes for all intermediate data structures per state x is given by [KGK19a]:

$$|Y_{\max}| := \max\left(|Y(\lambda_1)|, \ldots, \left|Y\left(\lambda_{|\lambda|}\right)\right|\right) \tag{10.7}$$

$$\text{intermediate}(x) := |Y_{\max}| \cdot (\text{size(R entry)} + 8 \text{ bytes}), \tag{10.8}$$

assuming that we assign all variables sequentially (see Section 10.2) and can reuse the buffer if it supports assigning the variable with the most possibilities ($|Y_{\max}|$). If this is not intended and all variables should be assigned concurrently, the memory consumption will increase to

$$\text{intermediate}(x) := \sum_{i=1}^{|\lambda|} |Y(\lambda_i)| \cdot (\text{size(R entry)} + 8 \text{ bytes}). \tag{10.9}$$

10.2.3 Algorithms

Algorithm 19 shows the actual successor generation GPU kernel designed for specialization [KGK19a]. The algorithm is our main contribution and realizes all the explained concepts earlier (see also Section 10.2.1). It takes a current state x, a given variable type λ^T, and a compatible heuristic H designed for the specified variable type. Note that λ^T is a meta-programming input, not an actual dynamic input variable [Kös+14c]. Its intended use case is to inline all H-related functions into the kernel, thus creating one kernel variant per variable type. This also affects selection of active variables from the parent optimization system, this algorithm would be used in (line 3).

The first steps in the algorithm are to perform shared-memory allocations (line 1). This includes space for sharing aggregator values and storing segment lookup information. In practice, the amount of shared memory required is usually determined at solve runtime and is therefore, realized via dynamic shared memory [NVI23a].

The next steps involve interpreting the successor value (see Figure 10.18) as a random seed used to drive the successor generation (line 2). Next, we enter the sequential variable-assignment loop that starts by resolving the number of possible assignments for the variable being assigned (line 4). The following steps (lines 5–7) involve the computation of all ratings, aggregating high-level information across all ratings, and mapping all the ratings. Note that each thread in the group will participate in these operations to exploit maximum parallelism and ensure high occupancy of all processing units. Details of these operations are shown in Algorithm 20 and Algorithm 21.

Lines 8–15 handle the case where no assignment could be determined. This is the case when the accumulated prefix sum does not result in a sum greater than zero. In this case, only the first thread of the group executes the required optimizer notification in terms of deactivating the variable. In any case, all threads in the group wait until the first thread has committed its changes (line 13) before continuing with the next variable assignment. Depending on the domain, a slight modification of the algorithm may be required at this point to potentially break the assignment loop if the optimizer has entered an unrecoverable state (see Section 10.2).

If an assignment is possible (lines 16–30), the initially required step is choosing a random value in the appropriate range (see Figure 10.16 and Equation (10.5)). This makes use of the loaded seed of the random-number generator and provides an updated seed for the next iteration. Based on Figure 10.17, we look for the matching segment using Algorithm 22 in line 17.

As discussed before, only the first warp is used to determine the actual variable assignment by testing all values in the segment interval (lines 18–26). Each lane of the warp tests a single offset of the determined segment using the cached prefix sums. It is sufficient to use the first warp, since the size of a segment is equal to the warp size.

Algorithm 19: Our fast successor generation on GPUs [KGK19a]

Input: State x, current variable type λ^T, heuristic H for the current type λ^T

```
   /* Shared memory allocations here                                  */
1  ...;
2  randomSeed := x.SuccessorValue;
3  foreach λ_i ∈ active variables of type λ^T do
4  |  |Y(λ_i)| := x.GetNumberOfPossibilities(H, λ_i);
   |  /* Algorithm 20                                                */
5  |  a := RateAllPossibilities(x, H, λ_i);
   |  /* Algorithm 21                                                */
6  |  MapAllRatings(x, H, λ_i, a);
   |  /* Check for a possible assignment                             */
7  |  rating := x.PrefixSumStorage[last segment];
8  |  if rating < 1 then
9  |  |  if group index = 0 then
   |  |  |  /* There is no possible assignment                      */
10 |  |  |  x.CouldNotAssign(λ_i);
11 |  |  |  Deactivate(λ_i);
12 |  |  end
   |  |  /* Wait for all threads to continue processing with
   |  |     the next variable                                       */
13 |  |  group barrier;
14 |  |  continue;
15 |  end
   |  /* Get random decision value v                                */
16 |  v, randomSeed := DrawRandom(randomSeed, [0, ..., rating − 1]);
   |  /* Algorithm 22                                                */
17 |  segment := FindMatchingSegment(x, λ_i, v);
   |  /* Select possibility based on v                              */
18 |  if warp index = 0 then
19 |  |  k := segment * warp size + group index;
20 |  |  leftSum := x.PrefixSumStorage[k - 1];
21 |  |  rightSum := x.PrefixSumStorage[k];
22 |  |  if leftSum ≤ v ∧ rightSum < v then
23 |  |  |  x.Assign(λ_i, possibility given by v);
24 |  |  |  Deactivate(λ_i);
25 |  |  end
26 |  end
27 |  group barrier;
28 end
   /* Update successor value                                         */
29 if group index = 0 then
30 |  x.SuccessorValue := randomSeed;
31 end
```

All remaining threads immediately encounter the following barrier (line 27) and wait for the assignment process to complete. By analogy to the case where no assignment is possible, the design guarantees that only a single thread in the first warp performs the assignment step (lines 23–24). The required operations are to assign the variable to the determined possibility and to disable the variable to skip further assignments. Finally, after all active variables of this kind have been assigned, we update the successor value in global memory to commit the state of the random number generator.

Algorithm 20: RateAllPossibilities for Algorithm 19 [KGK19a]

```
   Input: State x, local heuristic H, variable λ_i
1  a := H.CreateAggregator();
   /* Loop over all assignment possibilities, rate them, and
      store all ratings and use the aggregator                  */
2  for j := group index to |Y(λ_i)| step by group size do
3  |  R_i := H.ComputeRating(x, λ_i, j);
4  |  x.RatingStorage[j] := R_i;
5  |  a.Aggregate(R_i);
6  end
   /* Wait for group threads to reach this point                */
7  group barrier;
   /* Reduce all register-based aggregators and broadcast
      results to all threads in the group                       */
8  return group reduce (a);
```

Algorithm 21: MapAllRatings for Algorithm 19 [KGK19a]

```
   Input: State x, heuristic H, variable λ_i, reduced aggregator a
1  leftBoundary := 0;
   /* Loop over all ratings in the variable range               */
2  for j := group index to |Y(λ_i)| step by group size do
   |  /* Fetch rating from storage and map rating               */
3  |  storedRating := x.RatingStorage[j];
4  |  M_i := h.MapRating(storedRating, a);
   |  /* Compute lower prefix offset and update storage        */
5  |  prefixOffset := leftBoundary + group prefix sum (M_i);
6  |  x.PrefixSumStorage[j] := prefixOffset;
7  |  leftBoundary := group broadcast (prefixOffset, group size - 1);
8  |  if lane index = 0 then
9  |  |  SegmentLookup[ j / warp size ] := leftBoundary;
10 |  end
11 end
   /* Wait for group threads to reach this point                */
12 group barrier;
```

Both, Algorithm 20 and Algorithm 21 use group-stride loops to iterate over the number of assignment possibilities $|Y(\lambda_i)|$ of the current variable to assign λ_i (see also the *Implementation Details* section below). In the first case, we create a heuristic-specific (user defined) aggregator instance (line 1) and compute all rating values for each assignment possibility (line 3). The next step is to store ratings in the method-specific storage (line 4) and perform a thread-local aggregation (line 5), followed by a group wide reduction of all aggregator results in shared memory (line 8).

Algorithm 21 starts by loading previously computed ratings from our storage (line 3) and maps them to our intermediate rating format of type M (line 4). The idea is to continuously update the left boundary of the entire prefix-sum computation by performing group-wide prefix sums (line 5). To improve performance, we store the newly computed prefix offset in our temporary storage buffer (line 6) and update the left boundary for all threads via group-wide broadcasts from the last participating thread in the group (line 7). Then, each first lane in each participating warp stores the right boundary value (given by the last thread in the group) in the segment lookup (lines 8–10). Finally, we wait until all threads in the group have committed their changes.

Algorithm 22: FindMatchingSegment for Algorithm 19 [KGK19a]

```
   Input: State x, variable λ_i, value v
1  matchingSegmentIdx := −1;
   /* Iterate over segment lookup in shared memory          */
2  for j := group index + 1 to ⌈|Y(λ_i)| / warp size⌉ step by group size do
       /* Load boundaries from shared memory segment cache  */
3      leftBoundary := SegmentLookup[j − 1];
4      rightBoundary = SegmentLookup[j];
       /* Check whether we have found a match (see
          Equation (10.5))                                  */
5      match := leftBoundary ≤ v ∧ rightBoundary < v;
6      matchingSegmentIdx := group reduce (match ? j : −1);
7      if target segment matches then
8          return matchingSegmentIdx;
9      end
10 end
```

The last piece of our main algorithm is described by Algorithm 22. It is used to find a matching segment in our fast segment lookup table stored in shared memory that belongs to a drawn random decision value (Equation (10.5)). Since each segment represents a range of *warp-size* many outcomes, we perform a group-stride loop over all possibility segments (line 2, see also the *Implementation Details* section below). By loading the left and the right boundaries from the segment lookup (lines 3–4), we can immediately check if the segment is a match (line 5). Based on a group-wide reduction using a custom logic, keeping only positive values during reduction, we either get a single positive value or

no match at all. Once we find a matching segment, we can immediately break the loop for all threads (the group-wide reduction is assumed to return the same value for all threads). Note that a match is guaranteed by construction and that no further group barriers are required, since the group-wide reduction already requires synchronization between all threads.

Implementation Details

For the implementation of our main contribution, we chose C++ and CUDA for the entire implementation. The reason for that was to get an actual comparison between our C++ CPU version with automatic vectorization by the C++ compiler used for benchmarking purposes during the evaluation, while allowing code generation for the SSE and AVX instruction sets. In this way, it was possible to share parts of the code between the two worlds. Free variables were tracked by self-designed bit sets, using hardware instructions to jump from an active bit to the next [NVI23a].

Instantiating specialized kernels (see Algorithm 19) was realized using C++ meta-programming techniques via templates [AG04]. We also enforced inlining of all heuristic methods to avoid nested calls within loops, which would impose a significant overhead. Moreover, we used warp shuffles, optimized prefix-sum, and reduction implementations [NVI14; KGK19a] to improve efficiency of our implementation. Kernel specialization yielded highly optimized code instances with significantly reduced divergence in terms of the code paths being taking.

As for random-number generation, we used XorShift$\star$-based algorithms in practical applications of this method [KGK19a]. The advantage of these algorithms is that they are fast to execute, have a small memory footprint, and have sufficient graduation in terms of value distributions. Note that we initialize all of these generators with seeds from the CPU side which had been emitted by more complex number generators.

Algorithm 21 and Algorithm 22 suffer from thread divergence, since their loop-trip counts is not the same for all threads in the loop. This can lead to deadlocks or undefined behavior on GPUs since we leverage group-wide operations within these loops. We conservatively adjusted the loop boundaries in our implementation to perform the same number of iterations for all threads. In a similar way, we adjusted potential out-of-bounds shared-memory accesses by padding the shared-memory segments.

10.2.4 Performance Evaluation

In analogy to previously presented evaluations in this theses, we did not evaluate our newly introduced algorithm to compute $N(x)$ using external benchmarks, which may affect the reproducibility and the significance of the evaluation (see also Section 10.1.3). Therefore, we focused on the successor-generation processor itself, using a well established heuristic: The *Manhattan distance*, which is often used for path-finding problems [KGK19a]. This conceptually results in a 2D grid on which we tried to find a best path using the heuristic mentioned before given by [KGK19a]:

$$d(p,q) = \sum_{i=1}^{n} |p_i - q_i|, \tag{10.10}$$

were p and q are two points on our imaginary grid to compute the distance heuristic d for. The overall objective was to move a point p until we reached a target position on the grid. To do this, we moved our point p to a neighboring cell guided by our Manhattan-based distance heuristic d. In this setting, all neighboring cells of a single cell were candidate successor locations for a given point. They essentially acted as successor optimization states that could be explored in each heuristic optimization step.

In order to create reasonable workloads, we planned to move multiple points for which we intended to compute the shortest path to a set of given target positions. Each point was seen as a single variable λ_i that needed an assignment in each step. A step in this framework meant that each point is moved once to a different adjacent location on our evaluation grid. For evaluation purposes, we also chose $|\lambda| \in \{8, 32\}$ to analyze the impact of a relatively small number of variables on our runtime behavior [KGK19a]. To evaluate the impact of a small number of variables on our runtime behavior, we chose $|\lambda| \in \{8, 32\}$ [KGK19a]. Although the intended application scenarios were large-scale optimization problems involving thousands of variables, using a few variables gave us excellent insights about potential performance gains: This settings represented the *worst case of our method*, as we added a certain overhead to compute successor information per state. Theoretically, the worst case would be a single variable which just needed to differentiate between two possible options. However, we considered such a setting to be out of scope for our method since such a theoretical example was beyond our intended target application scenarios.

The number of possible assignments per variable λ_i per step ($|V_i|$) is given by the number of neighboring cells per point [KGK19a]:

$$|V_i| = j \cdot j - 1, \tag{10.11}$$

where j is the number of adjacent cells in a single dimension. To create differently challenging scenarios, we set j to be ≥ 35, even though the previously presented 2D problem required only 8 neighboring cells. However, this gave us

the opportunity to simulate larger workloads while computing the Manhattan distance separately for each assignment possibility V_i. We chose $j \in \{35, 67, 99\}$ to create workloads inspired by practical applications we encountered in several projects [KGK19a] (see also Part III). Before the actual variable assignment, the different selection possibilities V_i were mapped to a value $\in [0, \ldots, 8]$ in all cases to move our points to an existing neighboring cell. To have even more control over the computational load during heuristic evaluation, a load factor l was introduced. It controlled the number of repetitive Manhattan-distance computations in a loop that combined all results to make sure that the compiler cannot fold/eliminate the loops. The baseline value for all benchmarks performed was chosen to be $l = 1$. This again represented the worst-case in the scope of our evaluation scenario, as the computational load per possibility was extremely low. Since other common application scenarios of our method lie in the field of AI-driven heuristics involving chained matrix multiplications [Gel+12; KGK19a], we varied the computational load to investigate the scaling behavior of our method.

Similarly, we chose $|X|$ (the number of states) to be either 1024 or 4096, since extremely small workloads were out of scope of our application [KGK19a]. It reflected certain real-world characteristics of use cases we encountered, in which the number of assignment possibilities per variable was significantly smaller than the number of states ($|V_i| << |X|$) [KGK19a]. This usually avoided emerging of duplicate states during optimization (see also Section 10.2).

For the evaluation systems, we used two CPUs and two GPUs to compare our GPU-based implementation with an optimized CPU implementation. The CPUs used were the Intel Core i9 7940X (14 cores supporting 28 parallel threads) and the more recent AMD Ryzen 2700X (8 cores supporting 16 parallel threads)[3]. The GPUs used were two NVIDIA GPUs with different compute capabilities [NVI23a]: GeForce GTX Titan X and GeForce GTX 1080 Ti [KGK19a][4]. Since AMD's CPU was significantly slower than all other benchmark systems used, we present all benchmarks as speedups compared to the Ryzen 2700X (making the AMD system the baseline).

Figure 10.20 shows the first evaluation scenario involving 8 variables per state in the presence of 1024 states. As long as the problem was small (considering only 1224 assignment possibilities per variable), Intel's CPU was still pretty close. The speedup possible to achieve on our evaluation GPUs ranged from 1.6× on the GTX Titan X and 2.1× on the GTX 1080 Ti compared to the AMD CPU. In relation to the Intel CPU, we could only achieve a speedup of 1.5× on the GTX 1080 Ti.

[3]Note that both CPUs were well-known at the time of the publication. Therefore, we retained these measurements to present the original results. However, it may be of interest to reevaluate our method on more recent CPUs in future work (see Chapter 14).

[4]Note that both GPUs were state-of-the-art at the time of the publication. Therefore, we retained these measurements to present the original results. However, it is of interest to reevaluate our method on more recent GPUs in future work (see Chapter 14).

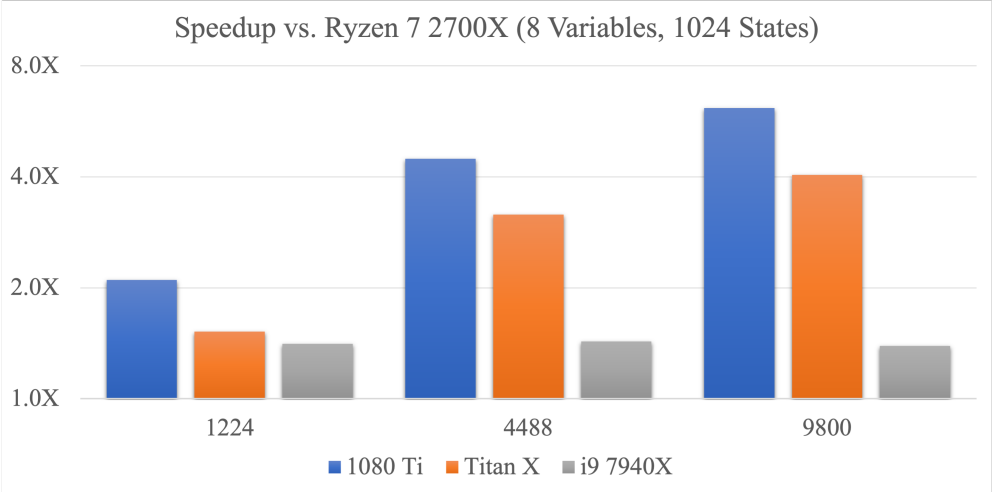

Figure 10.20: Performance comparison of our successor-generation algorithm using 8 variables and 1024 states while involving three value-assignment ranges $V_j \in \{1224, 4488, 9800\}$ (log2 scaling, higher is better, based on performance numbers published in [KGK19a]). Note that each variable is assigned sequentially one after the other.

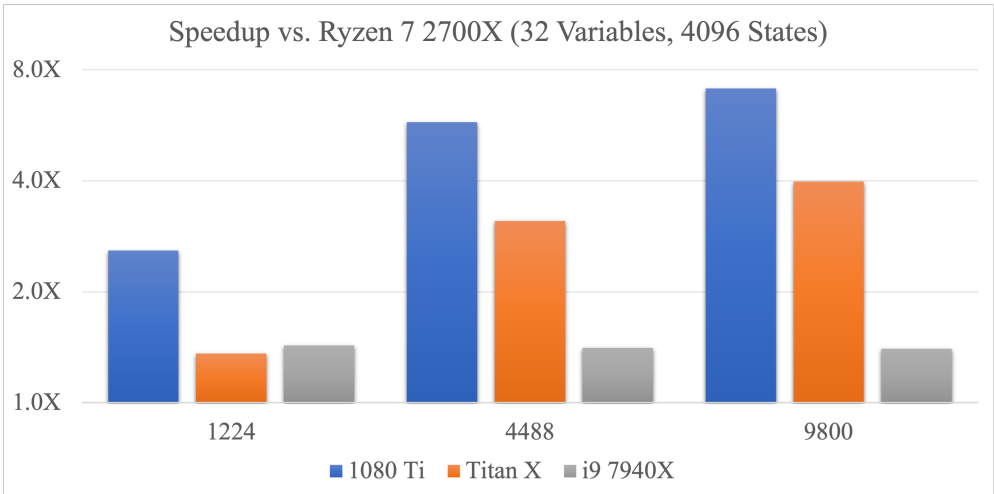

Figure 10.21: Performance comparison of our successor-generation algorithm using 32 variables and 4096 states while involving three value-assignment ranges $V_j \in \{1224, 4488, 9800\}$ (log2 scaling, higher is better, based on performance numbers published in [KGK19a]). Note that each variable is assigned sequentially one after the other.

This is due to the fact that the problem size is too small to exploit the parallel processing capabilities of our algorithm. Therefore, it is not possible to fully compensate for the additional overhead we introduced. Increasing the number of assignment possibilities in this scenario showed that the Intel CPU did not gain any speedup in comparison to the AMD CPU. We observed this behavior because the Intel CPU already reached their maximum utilization and could not scale to additional assignment possibilities. Fortunately, our algorithm scales very well on both GPUs and achieved speedups of 3.2× to 4× on the GTX Titan X and 4.5× to 6.1× on the GTX 1080 Ti [KGK19c]. However, we did not observe linear scaling as all variables were still sequentially assigned, which limited our full parallelization potential. The measured total execution times using 9800 assignment possibilities in this scenario where 31ms on the Ryzen 2700X, 21.3ms on the Core i9 7940X, 7.7ms on the GTX Titan X, and 5ms on the GTX 1080 Ti.

Increasing the number of sequentially assigned variables from 8 to 32 (see Figure 10.21) slightly improved speedups measured on the GTX 1080 Ti ranging from 2.6× (few assignment possibilities) to 7.1× (more assignment possibilities) [KGK19c]. Although we achieved a good overall scaling behavior by increasing the workload, our algorithm still suffered from linear variable assignments, which turned out to be a more serious problem on the GTX Titan X when combined with 1224 assignment possibilities. Here, the GTX Titan X was even slower than our Intel CPU. However, Intel's CPU did not scale at all.

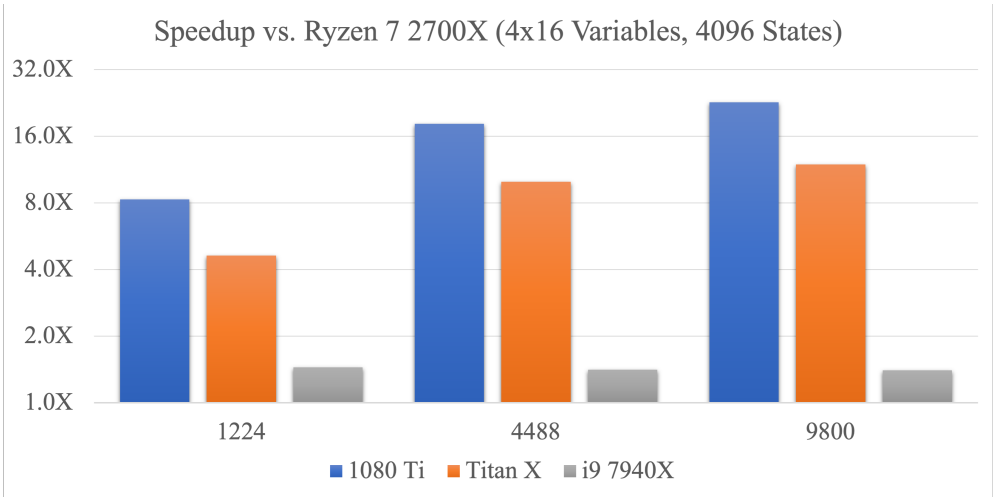

Figure 10.22: Performance comparison of our successor-generation algorithm using 4 groups of 16 variables each and 4096 states while involving three value-assignment ranges $V_j \in \{1224, 4488, 9800\}$ (log2 scaling, higher is better). Note that each variable within a group assigned sequentially one after the other. Variables in different groups are assigned in parallel since they do not depend on variables from another group.

To model common scenarios in which not all variables depend on each other, our algorithm also supports parallel assignment of variables. Figure 10.22 shows speedups measured by grouping 32 variables int 4 groups of 16 variables each. Internally, each variable group is assigned in parallel, while variables within each group are assigned sequentially[5].

More parallelization potential gave us a tremendous advantage in terms of scaling behavior and speedup in these benchmarks. Our measured speedups ranged from 4.6× to 11.9× on the GTX Titan X and from 8.3× to 23.7× on the GTX 1080 Ti compared to our base line. As before, the Core i9 did not scale when increasing the problem size. Note that the performance increased between 4488 assignment possibilities and 9800 was considerably less than before as we reached the maximum occupancy of both GPUs.

Increasing the number of states to 16384 revealed further huge speedups (see Figure 10.23). The speedups ranged from 11.6× to 30× on the GTX Titan X and from 24× to 68× on the GTX 1080 Ti compared to our base line. Again, the Intel processor did not scale at all when increasing the problem size. In analogy to the previous measurements, we did not see a considerable performance improvement when increasing the number of possibilities from 4488 to 9800.

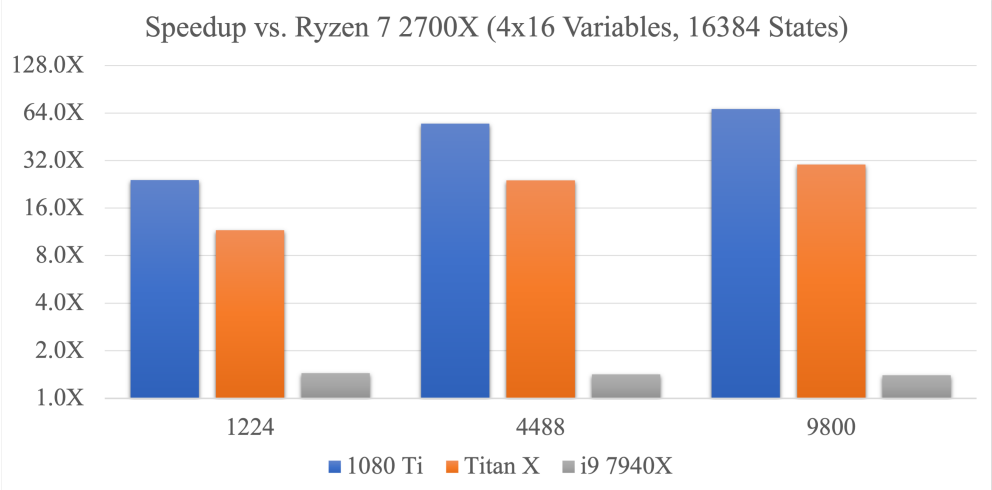

Figure 10.23: Performance comparison of our successor-generation algorithm using 4 groups of 16 variables each and 16384 states while involving three value-assignment ranges $V_j \in \{1224, 4488, 9800\}$ (log2 scaling, higher is better). Note that each variable within a group assigned sequentially one after the other. Variables in different groups are assigned in parallel since they do not depend on variables from another group.

[5]We slightly modified our surrounding optimizer implementation to support these use cases for this benchmark. Note that the Algorithm 19 we presented is designed to support parallel variable assignments, but custom grouping of variables was not shown.

CHAPTER 11

CONCLUSION

In 10.1 and 10.2, we presented contributions to the field of heuristics-based optimization systems. Here, we tackled two challenging problems from this domain: maintaining optimization states across multiple optimizer iterations and generating successor states. Section 10.1 described our high-level architectural solution to the state-maintenance problem (referred to as tracking and reconstruction in the context of our method). Using an on-the-fly reconstruction process based on highly compressed historical information, we were able to reduce memory consumption by orders of magnitude during an optimization process. Although reconstruction adds an overhead compared to keeping states in GPU memory (slowdowns of 3.5×), the actually measured slowdown was nonexistent compared to copying entire states from host CPU memory to the GPU for processing (speedups of 6.2× to 7.2×). Due to more limited memory sizes on GPUs these days [NVI23a], it is still often necessary to fetch data back to the CPU before running out of memory on an accelerator. As for the actual performance of state tracking during the optimization process, we conceptually differentiated between evaluation and recovery phases. The evaluation phase relies on a GPU-friendly form of generating successor states inside an optimization system, which should be explored afterwards using simulation logic (e.g., approaches from Part 1). Our fill-rate concept helped significantly to improve the overall GPU utilization and achieved speedups of 1.85× to 7.2×, depending on the evaluation scenario.

The most important contribution in the scope this thesis, is our approach, presented in Section 10.2, to efficiently generate successor states in a scalable manner solely the GPU. In contrast to domain-specific approaches from related work that used GPUs for local neighborhood exploration, our method is completely domain independent. Since it was designed with parallel processing (specifically GPU hardware) in mind, it uses specially designed temporary data structures materialized in both shared and global memory to make it as

efficient as possible. The multi-staged processing concept allows generic rating functions of different variable assignments, reasoning about aggregated rating information, and importance-based assignment sampling. Data stored in shared memory is primarily used to realize lookup tables to improve sampling performance, while data stored in global memory is used to cache computed ratings. As accesses are coherent and massively parallel, recomputing of different possibilities is often too expensive compared to storing and loading data. This is especially true for large-scale neighborhood searches involving expensive neural-network-based heuristics to realize the rating processes. Comparing our generic successor-generation algorithm to optimized CPU implementations, we measured speedups of 4.6× to 23.7× on smaller problem instances with just 4096 states in total. Increasing the number of states to more realistic problems resulted speedups ranging of 24× to 68×.

Answering RQ2 Our high-level solution for efficiently realizing tracking and reconstruction of states on the GPU enables optimizers to reduce memory consumption, improve the utilization of available resources, and achieve significant speedups over existing methods. Our method for efficiently generating successor states for arbitrary domains on the GPU greatly relaxes previous constraints on the size of the local neighborhood search space to be explored. This contributes to answering *RQ*2, as we have shown how our approach can achieve significant speedups over existing methods through shorter execution time, lower memory consumption, and better scalability (see also 13).

Part III

Engineering & Project Contributions, Conclusion and Future Work

CHAPTER 12

ENGINEERING & PROJECT CONTRIBUTIONS

Parts I and II focused on answering the central research questions of the thesis. At the same time, significant contributions were made to research and industry projects at the German Research Center for Artificial Intelligence, which include engineering contributions in particular. The theories, algorithms, and implementation details presented were used in the context of four major research projects (e.g, [Kon16; Kon19]) and four large industry projects at the time of publishing the individual papers [1]. A major goal within these projects was the real-time solution of large-scale optimization problems from the field of *Industrie 4.0*. Real-time here referred to a time horizon of just 40 milliseconds up to 30 seconds, depending on the problem size and application scenario. Besides these large projects, a set of smaller research projects from the fields of *Industrie 4.0*-related and *scientific visualization*-based domains were also completed using methods presented in this thesis.

To solve these real-world optimization problems using the acceleration methods from this thesis, we [2] built a completely new optimization framework, called *Any-Time Component Solver* (*ATCS*). The term any-time solver refers to the fact that the optimizer is able to stop at an arbitrary time and return the best solution found up to that time. This feature was essential to made it useful for practical problems: Depending on the optimization problem and the domain in which the system was used, the time horizon in which a solution had to be delivered varied depending on external and unpredictable influences. Such external influences refer to varying resource available, like machine-related interruptions, resource shortages, and customer order updates.

[1] The name of the industry projects are elided to not refer to any particular company.

[2] The thesis author's research development team that was founded to leverage the theory and algorithms presented in this thesis in practice.

12.1 Publications

The following list summarizes all contribution-relevant publications of this part of the thesis. Furthermore, contributions of the contributing author (*CA*) and all other authors (*CoA*) are explicitly listed. This helps to clearly separate own work that may be used in the dissertation from other contributions.

[KGK20a]	Marcel Köster et al. "High- Performance Simulations on GPUs Using Adaptive Time Steps." In: *20th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2020).* Springer, 2020
CA	Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking
CoA	Feedback on the paper, paper refinement, implementation and benchmarking
[KGK20b]	Marcel Köster et al. "Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction." In: *International Journal of Parallel Programming* (2020)
CA	Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking
CoA	Feedback on the paper, paper refinement, implementation and benchmarking
[KGK19a]⋆	Marcel Köster et al. "FANG: Fast and Efficient Successor-State Generation for Heuristic Optimization on GPUs." In: *19th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2019).* Springer, 2019
CA	Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking
CoA	Feedback on the paper, paper refinement, implementation and benchmarking

[KGK19c]	Marcel Köster et al. “Parallel Tracking and Reconstruction of States in Heuristic Optimization Systems on GPUs.” In: *Parallel and Distributed Computing, Applications and Technologies (PDCAT-2019)*. IEEE, 2019
CA	Idea generation, algorithm design, conceptual work on the paper, the diagrams and the use cases, implementation and benchmarking
CoA	Feedback on the paper, paper refinement

12.2 Generation of Specialized Optimizers

As mentioned in Parts I and II, we often leveraged specialization techniques for realizing our benchmarks. Specialization can be achieved through meta programming techniques and/or compiler support to generate code [Kös+14a; Kös+14c; Mem+14; KGK20a]. We have primarily used meta-programming and jit-based specialization at runtime, which is also specifically supported in ILGPU [Kös23] via so called *dynamically specialized kernels*. The latter concept allows us to leverage partial evaluation [Kös+14a] to aggressively optimize GPU kernels during runtime of the program [KGK19b; KGK20a]. This, in turn, is especially important for eliminating boundary checks, fully unrolling loops, and removing temporary local memory allocations that would otherwise have been required. As part of our engineering contributions to satisfy all of our partners' and customers' requirements, we built the *ATCS* framework on the .Net platform using the concept of *domain-specific language embedding* (DSL embedding, Section 12.2.1). Furthermore, we realized GPU acceleration using the ILGPU [Kös23] compiler, which has the ability to just-in-time compile .Net bytecode into GPU code while providing high- and low-level bindings to operate GPUs. This allowed us to target arbitrary platforms (e.g., *Linux* and *Windows*) by avoiding any kind of platform-dependent libraries while supporting a wide variety of different GPUs. Additionally, we could leverage existing language frontends (e.g, the C# compiler) to handle compilation of user-defined input code, while making use of the convenient development tools available for .Net.

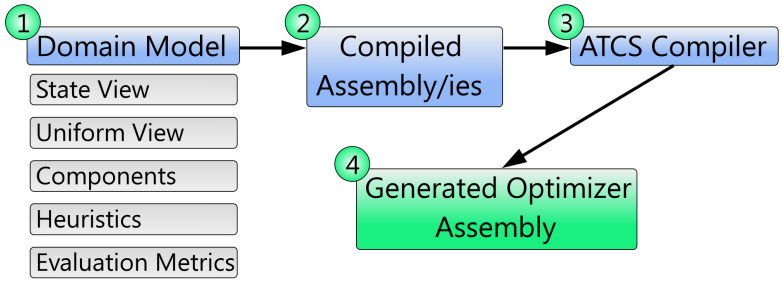

Figure 12.1: ATCS compilation workflow to generate specialized optimizers. First, the domain model is developed by a user of our framework in a .Net-compatible programming language (1). After compiling the model with its appropriate frontend compilers, we receive a set of compiled assemblies (or just a singe assembly in the most simplistic case possible, 2). Afterwards, we use our ATCS compiler that loads the pre-compiled domain model, disassembles it, and converts it into its own intermediate representation for analysis, optimization, and code-generation purposes (3). Finally, the output is a specialized optimizer assembly that contains all required kernels and search strategies to solve optimization problems compatible with the input domain (4).

The ability to integrate with existing development tools, combined with our ability to run the entire custom optimization model in a provided debug mode, allowed users to conveniently run through their components and heuristics using a CPU emulation layer. This greatly improved the development of optimization models in terms of required development time. Integrating our optimization system with cloud-hosted services was also straight forward, since ASP.Net services run using .Net libraries.

The entire system was built around a specially designed (yet generic) optimization library (written in C#) that contains interfaces to integrate with our internal optimization engine (see Figure 12.1). This engine used all the concepts presented in this thesis to provide better scalability and performance compared to other competitors (at the time of writing this thesis) [KGK19a; KGK19b; KGK19c; KGK20a; KGK20b]. In addition, the general idea was to automatically generate the entire optimizer using a compiler to benefit from domain knowledge to improve runtime performance (see below). In particular, this allowed us to determine optimal memory layouts, schedule components, and completely remove the overhead introduced by our software-defined abstractions. Furthermore, the real problem solved by our compiler was also to understand variable dependencies and their impact on certain data blocks of each state to enable proper parallelization.

In our context, developers define a *domain model* that contains an abstract description of all information required for each optimization state, as well as information that is shared by all states during optimization (referred to as *uniform* information). This model is implemented programmatically by providing an abstract interface definition, usually written in C# code. Moreover, the model contains an imperatively defined set of components operating on each state to realize domain-specific simulation logic (e.g., game rules). Each component follows the design principle of a time-adaptive generic component presented in Listing 6.4 that is generic with respect to the actual data structures used by leveraging view abstractions (see Chapter 6).

To create a working domain model, we also rely on provided evaluation metrics to evaluate states within out optimization system with respect to the optimization goal. These metrics are also provided by imperative interface implementations, ultimately in the form of .Net-compatible byte code. The last piece of information we need is a set of heuristics defining the strategy of how to explore the search space of our states using local heuristics (see also Section 10.2). In our real-world applications, such heuristics were either defined manually based on domain-expert knowledge or determined automatically by machine learning. Finally, it is important to note that a generated optimizer is able to optimize its own performance by monitoring execution times and utilization based on problem instances being solved. It uses the profile-based insights to dynamically specialize kernels and functions, while relying on just-in-time compilation to further improve runtime performance.

12.2.1 DSL Embedding

The term DSL refers to *domain-specific language* and indicates that it focuses on a specific domain or a set of domains, unlike general-purpose programming languages (like C#, C++, etc.) [Dan+14; Mem+14; Kös+14a]. This allows developers to benefit from expressing domain-specific knowledge directly without the need to model all specifics first in a general purpose programming language.

There are different ways on how to realize a DSL like defining a new language or embed a DSL into another (often called *host*) language [Kös+14a]. We used the general approach of *DSL embedding*, by choosing the .Net ecosystem as our host environment. This decision gave us the advantage that ATCS built upon an existing ecosystem of tools and runtimes for a huge variety of different platforms [3]. We also benefited from extremely fast compile times and GPU hardware support (via ILGPU). In addition, we could exploit runtime specialization via the .Net JIT and ILGPU to create optimized code at runtime [4]. This was extremely valuable as optimization problem instances tend to vary from invocation to invocation of the optimization system.

Applying similar techniques at compile time would have required fine-grained information about the actual problem instances. Consequently, we would have had to recompile the domain model for each problem instance or generate a considerable amount of pre-specialized instances to chose from at runtime. Similarly, GPU kernels would have had to be precompiled for all potential target platforms. Because we relied on runtime specialization and JIT compilation, we were able to avoid excessive pre-specialization and pre-compilation in general. However, the ATCS compiler could also incorporate profiling based performance information into its code-generation pipeline. This allowed to pre-optimize generated code in terms of general memory layouts in addition to runtime specialization, which happens in all cases.

Embedding into .Net DSL embedding into .Net was realized leveraging abstract interfaces. Interfaces were used to define the actual optimization domain in terms of the information that needs to be tracked per state. Listing 12.1 shows a simplified version of our provided state interface. It describes the basic properties of each optimization state and depends on a generic parameter *TEvaluationType*. This type parameter allows developers to specify which data type to use when evaluating states with respect to other states during evaluation (see also Section 10.1 and Section 10.2). We used the suffix *View*

[3] As mentioned before, this also included Azure- and self-hosted web services, desktop applications, background services, mobile applications, and even web browsers via webassembly compilation.

[4] Using ILGPU also allowed us to benefit from CPU-based GPU emulation. This in turn enabled debugging of the whole optimizer, the user-defined heuristics and simulation components conveniently on the CPU. Note that debugging could be realized using standard .Net debugging tools.

to indicate that it is an abstraction to access data stored somewhere in some format to be determined by ATCS. This terminology was influenced by the ILGPU naming scheme to access GPU memory via *ArrayViews* [Kös23].

Listing 12.1: Simplified state interface in C# code

```
interface IStateView<TEvaluationType>
    where TEvaluationType : struct, IComparable<TEvaluationType>
{
    /// <summary>
    /// Returns the unique state id to differentiate between individual
        states.
    /// </summary>
    long StateId { get; }

    /// <summary>
    /// Returns the current evaluation result of this state (if any).
    /// </summary>
    TEvaluationType? EvaluationResult { get; }

    /// ...
}
```

To define a custom state representation, developers specify their own abstract state interface while inheriting from our *IStateView* interface. Listing 12.2 shows a simplistic state representation storing a 1D "array" of agents (to be simulated) in this sample. Note that the type *IView1D* is an ATCS provided abstraction to avoid direct accesses to memory in the domain model. Actual views implementations are provided by ATCS when compiling the domain model (see Section 12.2.2). The actual length of this view per state will be determined at optimizer runtime as soon as an optimization problem instance is loaded. Moreover, this sample also specifies a static readonly property *Obstacles* returning a readonly 1D view. This static property tells ATCS that there is static information in form of *uniforms* accessible by all states. Uniforms are constants with respect to the optimization process in ATCS land but can be defined before solving an optimization problem.

Listing 12.2: Sample domain using several Agent instances per state

```
interface ICustomStateView : IStateView<Float32Domain>
{
    /// <summary>
    /// Defines a uniform read-only view accessible by all states.
    /// </summary>
    static IReadOnlyView1D<Obstacle> Obstacles { get; }

    /// <summary>
    /// Returns a view to access multiple agents in 1D.
    /// </summary>
    IView1D<Agent> Agents { get; }
}
```

12.2.2 Memory Layouts and Specialization

As outlined in the previous section, state descriptions, components, and heuristics were based on interfaces. Once the user-defined model is realized via our provided optimization-domain-specific abstractions, ATCS can analyze all parts of the input model[5]. Note that some performance optimizations will delayed to be performed at *optimizer runtime* as soon as the optimization problem instance is known (see paragraph *Specialization*).

Memory Layouts A very important step to utilize memory controllers of GPUs properly (see also Section 2.4) is determining an optimal *memory layout*. The idea was to layout data in memory in such a way that memory accesses will be coalesced automatically. This is an optimization problem on its own, focussing on the placement and alignment of data in memory buffers.

Consider the user-defined structure shown in Listing 12.3 and an abstract view interface shown in Listing 12.4. The idea is that all components and heuristics provided by the user operate on such abstract interfaces. This concept allows us to provide specialized implementations automatically that are optimized for the problem being solved while taking the hardware abilities into account.

Listing 12.3: Sample structure of a simple agent supporting simple movement from current 2D positions to 2D goal positions while taking a remaining amount of time in milliseconds into account.

```
1 struct Agent
2 {
3     Vector2D CurrentPosition;
4     Vector2D GoalPosition;
5     int RemainingTime;
6 }
```

Listing 12.4: Abstract view to access Agent structures from Listing 12.3.

```
1 interface IAgentView
2 {
3     /// <summary>
4     /// Gets or sets the ith Agent structure instance.
5     /// </summary>
6     Agent this[int index] { get; set; }
7 }
```

Based on guidelines to optimize memory layouts, there are several potential candidate solutions (see Section 6.2 and [NVI23a; Kös23]). Assume that fields of this data structure will be accessed in parallel and by multiple components. Having this information at hand, a structure-of-array (SOA) memory layout may be well suited given the fact that memory access happen in parallel. This allows us to create a specialized *IAgentView* implementation splitting all fields into 32-bit chunks shown in Listing 12.5.

[5] After the user model has been compiled into .Net bytecode using standard toolchains.

Listing 12.5: SOA layout for Listing 12.3 while implementing Listing 12.4.

```
struct SOAAgentView : IAgentView
{
    private View1D<float> current_x;
    private View1D<float> current_y;

    private View1D<float> goal_x;
    private View1D<float> goal_y;

    private View1D<int> remainingTime;

    public Agent this[int index]
    {
        get
        {
            // Load individual elements from our buffers and assemble the
            // agent data structure on the fly
            Vector2D current = new(current_x[index], current_y[index]);
            Vector2D goal = new(goal_x[index], goal_y[index]);
            return new Agent(current, goal, remainingTime[index]);
        }

        // The setter is omitted here
    }
}
```

Although the getter looks promising at first sight, we may encounter different uses of this property in our input domain model. Consider the case where some parts read the full data structure while others read only parts of it. Performing all five loads naively every time will degrade performance in cases where we only partially access loaded data. Since we built upon ILGPU, it was able to perform dead-load optimizations aggressively at kernel compile time, removing not required loads. However, this might still not eliminate all unnecessary loads due to potential read-write dependencies within a single generated kernel (see paragraph *Specialization*). Even worse, generating generic setters that write all fields into memory will not leave any room for further optimizations. This is because ILGPU was not be able to proof that deleting writes into main memory may not violate the semantics of the input program.

In order to generate efficient views using optimized data accesses, ATCS leveraged domain information from the input domain model. The ATCS compiler generated specialized view implementations for each involved component while minimizing loads and stores. Furthermore, it also addressed a different performance issue: Depending on the internal structure and complexity of individual components, processing all agents per state in parallel may not lead to optimal occupancy (in this sample). This is because users optimize many states in parallel and focus on the overall efficiency of investigating and evaluating all states in parallel.

To tackle this problem, ATCS supported profile-guided optimization to allow integration of benchmark results into the code-generation process. This actively influenced the decisions about the underlying memory layout and could lead to fusion of data views. We provided a greedy bin-packing algorithm

and a considerably more expensive constraint-based optimization algorithm to perform fusing of data structures in these cases. The general process always started by assuming a fine-grained SOA layout and fused views into combined densely-packed partial data structures one after another. In the case of our agent view interface, an implementation may store the *CurrentPosition* as an actual *Vector2D* while separating the goal fields as shown before.

Specialization We applied *specialization* in multiple stages and achieved optimization via generic types, custom generated types/methods, and inlining. Generic types allowed us to define an implementation type at a later stage by binding the generic type parameters to generated types. This also realized a separation of concerns that domain developers were not exposed to optimizer and implementation internals. This enabled us to use aggressively inline methods after type specialization to completely remove the overhead of our high-level abstractions.

Specialization in our case happened at domain-model compile time within ATCS and at runtime of the generated optimizer. Compile-time specialization involved creating specialized structures, methods, and whole kernels to link the ATCS optimizer algorithms with the user-domain model code. This process included generating memory-access views (as mentioned in the paragraph on *Memory Layouts*), all GPU kernels, as well as wrapper code to initialize all memory buffers structures and load all kernels at optimizer runtime.

At optimizer runtime, the .Net JIT performed actual type specializations that glued the input-domain model with the generated data structures of ATCS. It was achieved by binding generic type parameters to implementation types provided in the generated code. Moreover, even the ATCS optimizer runtime depended on generic type parameters to enforce specialization (to remove abstraction overhead). We also leveraged specific *method-implementation attributes* (the *MethodImplAttribute* [Lid02]) on most methods to trigger aggressive inlining behavior or forced enabling aggressive optimizations in the .NET JIT compiler.

In addition, we made use of ILGPU's dynamic specialization feature to specialize GPU kernels dynamically at program runtime. This feature allowed passing values to selected kernel parameters and enforced partial evaluation-like behavior of kernels by embedding the given values as compile-time-known constants. By doing so, we automatically triggered additional optimization passes of ILGPU and created optimized kernels. For instance, this allowed us to fold if-conditions and unroll loops based on actual problem instances being solved by the optimizer.

CHAPTER 13

CONCLUSION

This chapter concisely summarizes the whole thesis while focusing on the aim of this thesis. The aim was primarily given by answering our two main research questions (see Section 1.3). As mentioned in Section 1.3, neither question can be answered exhaustively. Both questions were meant to guide our research in terms of contributions intended to be made. As mentioned in Section 1.3, neither question can be answered exhaustively. Both questions should guide our research in terms of contributions to be made.

Part 1 focused on answering RQ_1 and contributed several domain-specific and domain-independent methods (Part 1 Conclusion, Chapter 7). Our methods leveraged currently available GPU capabilities to overcome runtime performance, memory consumption, and scalability limitations in the parallel simulation domain. Considering our contributions, we achieved our research goal of contributing to the answer of RQ_1.

The answer to RQ_2 was tackled by Part 2. We presented a high-level concept to realize GPU-aware state tracking and a game-changing scalable solution for generating successor states on the GPU for optimizers. Our contributions are compatible with arbitrary domains, as they do not rely on any domain knowledge. Most importantly, as in Part 1, we were able to overcome existing limitations in terms of scalability, memory consumption, and runtime behavior (Part 2 Conclusion, Chapter 11). In summary, our contributions in Part 2 also achieved our goal of answering RQ_2.

Beyond answering the main research questions, our approaches were incorporated into a new optimization system, which in turn was used in multiple research and industry projects (Chapter 12). This led to engineering contributions that were also achieved in collaboration with other research engineers. As part of various projects, other engineers gathered actual requirements and built optimization models based on our framework (Section 12.2).

To summarize, we achieved substantial performance enhancements compared to existing solution within our primary areas of focus.

CHAPTER 14

LIMITATIONS & FUTURE WORK

Naturally, our methods presented in this thesis have certain limitations. Starting with our iteration-adaptive fluid simulations, there are some implicit assumptions to ensure stability. Reconsider our adaptation models that assign different CL-information to each particle. Currently, the two proposed models rely on linear degradation of CL information. Introducing adaptation models that adjust CL in a non-linear way could lead to unstable simulations and potentially violate average-density information. An interesting idea for the future is to extend our adaptation method to whole PBD solvers instead of adaptively solving only a part of them (although this is the most expensive part).

In addition, discussing our newly introduced selection method for particle-based domains, we would like to analyze more datasets from various domains. Extending our investigations to more input scenarios will allow us to refine our mask-mapping approach. A potential drawback of our current approach is that certain parameters may need adjustment (e.g., the initially guessed smoothing length) when operating on arbitrary datasets. This limitation could be addressed by implementing an automated parameter-tuning algorithm, which might or might not utilize machine-learning methods. Moreover, enhancing precision using machine-learning based matching approaches trained on given selection masks may also prove beneficial.

Moving on to our contributions to generic simulations, the main drawback of our suggested memory layout is the overhead of our address-computations: Several additional arithmetic operations are necessary compared to simpler layouts. Although we did not see any considerable performance impacts, it is possible that memory-bound rules that rely on multiple IO instructions are negatively affected. However, the latest trend in the GPU space indicates an improvement in computational power, so these concerns may also become obsolete [NVI23a; KGK20a]. One conceptual enhancement is to avoid assigning a single warp to a particular state. Rethinking this design decision by

allowing arbitrary threads to operate on different states can further improve performance. However, this will also increase the overall complexity of our compaction logic and our memory-address computations.

Also in the simulation domain, adaptive time stepping may lead to deviations based on the choice interpolation function used. Simulation deviations refer to errors compared to non-adaptive simulation implementations. Parameters controlling the actual time-step maxima and those of interpolation functions must be fine-tuned depending on the domain to which they are applied in all cases. Since this is also a limitation of all related methods, especially generic ones, we argue that this is not a serious drawback of our method. In the future, it is a promising idea to explore more comprehensive (statically executed) component analyses, which could potentially lead to the development of advanced caching concepts that can be generated automatically.

Limitations also apply to the methods presented in the second part of this thesis. Our tracking and reconstruction approaches do not currently use advanced recovery methods. By advanced here, we refer to predictive methods that can anticipate specific recovery steps to improve overall utilization in cases where even our fill rate-based methods cannot guarantee maximum utilization. On the one hand, our recovery method can reduce memory consumption by orders of magnitude. On the other hand, the main disadvantage of our successor-generation algorithm is probably its high memory consumption. The method requires a few bytes per assignment possibility in each state processed in parallel to avoid caching already computed ratings.

In general, this is again not a severe limitation, since mappings can be recomputed to reduce memory requirements.. However, we have not yet seen an practical limitation on individual use cases. Given the ongoing trend to increase GPU computing performance, recalculations may even become cheaper in the future and making caching obsolete (see above). In addition, an automatically determined caching configuration (or even the introduction of partial caching) can be beneficial. This may be based on static or automatically trained heuristics. Also, since there is a trend to increase the size of fast on-chip shared memory, more excessive caching may also utilize additional space in shared memory by loading precomputed rating values.

The general direction for future research is to apply our findings to characteristic machine learning problems. Although we focused on heuristic optimization in general and implicitly included ML-driven optimization systems, we did not directly contribute to foundations of ML-based approaches. This domain in particular can benefit even more from our conceptual contributions in the future. Consequently, this is also a direction we intend to take in the future.

LIST OF ALGORITHMS

LIST OF FIGURES

BIBLIOGRAPHY

[ACK13] Omar Abdelkafi, Khalil Chebil, and Mahdi Khemakhem. "Parallel local search on GPU and CPU with OpenCL Language." In: Proceedings of the first international conference on Reasoning and Optimization in Information Systems. Sept. 2013.

[AG04] David Abrahams and Aleksey Gurtovoy. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional, 2004.

[Ada+07] Bart Adams et al. "Adaptively Sampled Particle Fluids." In: ACM Transactions on Graphics (2007).

[Aki+12] Nadir Akinci et al. "Versatile Rigid-Fluid Coupling for Incompressible SPH." In: ACM Transactions on Graphics (2012).

[AMD19] AMD. AMD Vega Instruction Set Architecture. 2019.

[Amd67] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference. 1967.

[B J+11] B. J. Ferdosi et al. "Comparison of density estimation methods for astronomical datasets." In: A&A (2011).

[BH86] Josh Barnes and Piet Hut. "A hierarchical O(N log N) force-calculation algorithm." In: Nature (1986).

[BT07] Markus Becker and Matthias Teschner. "Weakly compressible SPH for free surface flows." In: Symposium on Computer Animation. 2007.

[BOA09] Markus Billeter, Ola Olsson, and Ulf Assarsson. "Efficient Stream Compaction on Wide SIMD Many-Core Architectures." In: Association for Computing Machinery, 2009.

[BLS12] Kenneth Bodin, Claude Lacoursiere, and Martin Servin. "Constraint Fluids." In: IEEE Transactions on Visualization and Computer Graphics (2012).

[Bro+12] Cameron Browne et al. “A survey of Monte Carlo tree search methods.” In: IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI (2012).

[Cam+14a] Federico Campeotto et al. “A GPU Implementation of Large Neighborhood Search for Solving Constraint Optimization Problems.” In: Proceedings of the Twenty-first European Conference on Artificial Intelligence. 2014.

[Cam+14b] Federico Campeotto et al. “Exploring the Use of GPUs in Constraint Solving.” In: Practical Aspects of Declarative Languages. ACM, 2014.

[Car+10] Varis Carey et al. “Blockwise Adaptivity for Time Dependent Problems Based on Coarse Scale Adjoint Solutions.” In: SIAM Journal on Scientific Computing (2010).

[CWH08] Guillaume M. J. -B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. “Parallel Monte-Carlo Tree Search.” In: Computers and Games. 2008.

[DFK12] Florian Daiber, Eric Falk, and Antonio Krüger. “Balloon Selection revisited - Multi-touch Selection Techniques for Stereoscopic Data.” In: Proceedings of the International Conference on Advanced Visual Interfaces. 2012.

[Dai+14] Florian Daiber et al. “Interacting with 3D Content on Stereoscopic Displays.” In: PerDis. 2014.

[Dan+14] Piotr Danilewski et al. “Specialization through Dynamic Staging.” In: Proceedings of the 13th International Conference on Generative Programming: Concepts & Experiences (GPCE). ACM, 2014.

[DA12] Walter Dehnen and Hossam Aly. “Improving convergence in smoothed particle hydrodynamics simulations without pairing instability.” In: (2012).

[DG96] Mathieu Desbrun and Marie-Paule Gascuel. “Smoothed Particles: A new paradigm for animating highly deformable bodies.” In: Proceedings of EG Workshop on Animation and Simulation. 1996.

[FLL04] Filippo Focacci, Francois Laburthe, and Andrea Lodi. “Local Search and Constraint Programming.” In: Constraint and Integer Programming: Toward a Unified Methodology. 2004.

[FA11] W. W. L. Fung and T. M. Aamodt. “Thread block compaction for efficient SIMT control flow.” In: 2011 IEEE 17th International Symposium on High Performance Computer Architecture. 2011.

[GH13] Martin Gander and Laurence Halpern. “Techniques for Locally Adaptive Time Stepping Developed over the Last Two Decades.” In: Lecture Notes in Computational Science and Engineering (2013).

[Gar+11] V. M. Garcia et al. “An adaptive step size GPU ODE solver for simulating the electric cardiac activity.” In: 2011 Computing in Cardiology. 2011.

[Gel+12] Sylvain Gelly et al. “The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions.” In: Communications of the ACM (2012).

[GK14] Shreekant Ghorpade and Snehal Kamalapur. “Solution Level Parallelization of Local Search Metaheuristic Algorithm on GPU.” In: 2014.

[GM77] Robert A. Gingold and Joe J. Monaghan. “Smoothed Particle Hydrodynamics-Theory and application to nonspherical stars.” In: Notices of the Royal Astronomical Society (1977).

[GB14] Prashant Goswami and Christopher Batty. “Regional Time Stepping for SPH.” In: Eurographics 2014 - Short Papers. The Eurographics Association, 2014.

[GP11] Prashant Goswami and Renato Pajarola. “Time Adaptive Approximate SPH.” In: Workshop in Virtual Reality Interactions and Physical Simulation. 2011.

[Gre10] Simon Green. Particle Simulation using CUDA – Parallel Radix Sort. 2010.

[GM12] Kate Gregory and Ade Miller. C++ AMP: Accelerated Massive Parallelism with Microsoft Visual C++. Microsoft Press, 2012.

[GKK19] Julian Groß, Marcel Köster, and Antonio Krüger. “Fast and Efficient Nearest Neighbor Search for Particle Simulations.” In: Proceedings of the Conference on Computer Graphics & Visual Computing (CGCV-2019). The Eurographics Association, 2019.

[GKK20] Julian Groß, Marcel Köster, and Antonio Krüger. “CLAWS : Computational Load Balancing for Accelerated Neighbor Processing on GPUs using Warp Scheduling.” In: Proceedings of the Conference on Computer Graphics & Visual Computing (CGCV-2020). The Eurographics Association, 2020.

[Gro23] The Khronos Group. Open Standard For Parallel Programming Of Heterogeneous Systems. 2023. URL: https://www.khronos.org/opencl/ (visited on 05/06/2023).

[Heg+12] Hans-Christian Hege et al. “WYSIWYP: What You See Is What You Pick.” In: IEEE Transactions on Visualization and Computer Graphics (2012).

[Hob+09] Jared Hoberock et al. “Stream Compaction for Deferred Shading.” In: 2009.

[Hoe14] Rama Hoetzlein. Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids. GPU Technology Conference. 2014.

[HHK08] Woosuck Hong, Donald H. House, and John Keyser. “Adaptive Particles for Incompressible Fluid Simulation.” In: The Visual Computer (2008).

[HHK09] Woosuck Hong, Donald H. House, and John Keyser. “An Adaptive Sampling Approach to Incompressible Particle-Based Fluid.” In: Theory and Practice of Computer Graphics. 2009.

[HS13] Christopher J. Horvath and Barbara Solenthaler. Mass Preserving Multi-Scale SPH. Pixar Technical Memo 13-04, Pixar Animation Studios. 2013.

[Hug+13] D. M. Hughes et al. “InK-Compact: In-Kernel Stream Compaction and Its Application to Multi-Kernel Data Visualization on General-Purpose GPUs.” In: (2013).

[Hun07] Raphael Hunger. Floating Point Operations in Matrix-Vector Calculus. Tech. rep. Technische Universität München, 2007.

[Ihm+10] Markus Ihmsen et al. “Boundary Handling and Adaptive Timestepping for PCISPH.” In: VRIPHYS. 2010.

[Ihm+14] Markus Ihmsen et al. “Implicit Incompressible SPH.” In: Visualization and Computer Graphics, IEEE Transactions on (2014).

[Kay+10] David Kay et al. “Adaptive Time-Stepping for Incompressible Flow Part II: Navier–Stokes Equations.” In: SIAM Journal on Scientific Computing (2010).

[KSW99] Jin Koda, Yoshiaki Sofue, and Keiichi Wada. “How to Determine the Smoothing Length in Sph?” In: Astrophysics and Space Science Library. 1999.

[Kon19] BaSys4.0 Konsortium. BaSys 4.0 - Basissystem Industrie 4.0, Abschlussbericht: Berichtszeitraum: 1.7.2016-30.06.2019. Tech. rep. Robert Bosch GmbH, 2019. URL: https://www.tib.eu/de/suchen/id/TIBKAT%3A1690357495.

[Kon16] SmartF-IT Konsortium. BMBF-Verbundprojekt SmartF-IT - cyber-physische IT-Systeme zur Komplexitätsbeherrschung einer neuen Generation multiadaptiver Fabriken : Schlussbericht Konsortialleitung : 01.06.2013 − 31.08.2016. 2016. URL: https://www.tib.eu/de/suchen/id/TIBKAT%3A883941694.

[Kös13] Marcel Köster. “An Interactive Space Simulation For Media Facades.” Bachelor’s Thesis. Saarland University, 2013.

[Kös23] Marcel Köster. ILGPU JIT Compiler. 2023. URL: https://www.ilpug.net/ (visited on 05/06/2023).

[KGK19a] Marcel Köster, Julian Groß, and Antonio Krüger. "FANG: Fast and Efficient Successor-State Generation for Heuristic Optimization on GPUs." In: 19th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2019). Springer, 2019.

[KGK19b] Marcel Köster, Julian Groß, and Antonio Krüger. Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction. 12th International Symposium on High-Level Parallel Programming and Applications (HLPP-2019). 2019.

[KGK19c] Marcel Köster, Julian Groß, and Antonio Krüger. "Parallel Tracking and Reconstruction of States in Heuristic Optimization Systems on GPUs." In: Parallel and Distributed Computing, Applications and Technologies (PDCAT-2019). IEEE, 2019.

[KGK20a] Marcel Köster, Julian Groß, and Antonio Krüger. "High- Performance Simulations on GPUs Using Adaptive Time Steps." In: 20th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2020). Springer, 2020.

[KGK20b] Marcel Köster, Julian Groß, and Antonio Krüger. "Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction." In: International Journal of Parallel Programming (2020).

[KK16] Marcel Köster and Antonio Krüger. "Adaptive Position-Based Fluids: Improving Performance of Fluid Simulations for Real-Time Applications." In: International Journal of Computer Graphics & Animation (2016).

[KK18] Marcel Köster and Antonio Krüger. "Screen Space Particle Selection." In: Proceedings of the Conference on Computer Graphics & Visual Computing (CGCV-2018). The Eurographics Association, 2018.

[KSG15a] Marcel Köster, Michael Schmitz, and Sven Gehring. "An Interactive Planetary System for High-Resolution Media Facades." In: Proceedings of the International Symposium on Pervasive Displays. International Symposium on Pervasive Displays (PerDis-15), June 10-12, Saarbrücken, Germany. ACM, 2015.

[KSG15b] Marcel Köster, Michael Schmitz, and Sven Gehring. "Gravity Games - A Framework for Interactive Space Physics on Media Facades." In: Proceedings of the International Symposium on Pervasive Displays. ACM, 2015.

[Kös+14a] Marcel Köster et al. “Code Refinement of Stencil Codes.” In: Parallel Processing Letters (PPL) (2014).

[Kös+14b] Marcel Köster et al. “High-Performance Domain-Specific Languages for GPU Computing.” In: GPU Technology Conference (GTC-14) (2014).

[Kös+14c] Marcel Köster et al. “Platform-Specific Optimization and Mapping of Stencil Codes through Refinement.” In: Proceedings of the 1st International Workshop on High-Performance Stencil Computations (HiStencils-2014). 2014.

[Kös+15] Marcel Köster et al. “Asterodrome: Force-of-Gravity Simulations in an Interactive Media Theater.” In: Proceedings of the 14th International Conference on Entertainment Computing (ICEC-2015). Springer, 2015.

[Kri01] S. Krishnaprasad. “Uses and Abuses of Amdahl’s Law.” In: J. Comput. Sci. Coll. (2001).

[LGS09] Wladimir van der Laan, Simon Green, and Miguel Sainz. “Screen Space Fluid Rendering with Curvature Flow.” In: Proceedings of the Symposium on Interactive 3D Graphics and Games. 2009.

[LTL13] Yuet Lam, Kuen Tsoi, and Wayne Luk. “Parallel neighbourhood search on many-core platforms.” In: International Journal of Computational Science and Engineering (2013).

[LOL19] Tae Min Lee, Young Jin Oh, and In-Kwon Lee. Efficient Cloth Simulation using Miniature Cloth and Upscaling Deep Neural Networks. 2019.

[LKH15] Roland Leißa, Marcel Köster, and Sebastian Hack. “A Graph-Based Higher-Order Intermediate Representation.” In: Proceedings of the International Symposium on Code Generation and Optimization (CGO). ACM, 2015.

[Lew21] Christopher Lewin. “Swish: Neural Network Cloth Simulation on Madden NFL 21.” In: ACM SIGGRAPH 2021 Talks. ACM, 2021.

[LLK19] Junbang Liang, Ming C. Lin, and Vladlen Koltun. “Differentiable Cloth Simulation for Inverse Problems.” In: Proceedings of the 33rd International Conference on Neural Information Processing Systems. Curran Associates Inc., 2019.

[Lid02] Serge Lidin. Inside Microsoft .NET IL Assembler. Microsoft Press, 2002.

[LC87] William E. Lorensen and Harvey E. Cline. “Marching cubes: A high resolution 3D surface construction algorithm.” In: Computer Graphics (1987).

[Luc77] Leon B. Lucy. “A numerical approach to the testing of the fission hypothesis.” In: Astronomy Journal (1977).

[Luo+10] T. V. Luong et al. “A GPU-based iterated tabu search for solving the quadratic 3-dimensional assignment problem.” In: ACS/IEEE International Conference on Computer Systems and Applications (AICCSA). 2010.

[LMT10a] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. “Large Neighborhood Local Search Optimization on Graphics Processing Units.” In: Workshop on Large-Scale Parallel Processing (LSPP) in Conjunction with the International Parallel & Distributed Processing Symposium (IPDPS). 2010.

[LMT10b] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. “Neighborhood Structures for GPU-based Local Search Algorithms.” In: Parallel Processing Letters (2010).

[LSG19] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. “A Formal Analysis of the NVIDIA PTX Memory Consistency Model.” In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. 2019.

[MM13] Miles Macklin and Matthias Müller. “Position Based Fluids.” In: ACM Trans. Graph. (2013).

[Mac+14] Miles Macklin et al. “Unified Particle Physics for Real-time Applications.” In: ACM Trans. Graph. (2014).

[MWG18] Matthias Mayr, Wolfgang Wall, and Michael Gee. “Adaptive time stepping for fluid-structure interaction solvers.” In: Finite Elements in Analysis and Design (2018).

[Mel+11] Nouredine Melab et al. “ParadisEO-MO-GPU: a framework for parallel GPU-based local search metaheuristics.” In: 11th International Work-Conference on Artificial Neural Networks. 2011.

[Mem+14] Richard Membarth et al. “Target-Specific Refinement of Multigrid Codes.” In: Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC-2014). IEEE, 2014.

[Min95] Mark R. Mine. Virtual Environment Interaction Techniques. Tech. rep. 1995.

[Mon92] Joe J. Monaghan. “Smoothed particle hydrodynamics.” In: Annual Review of Astronomy and Astrophysics. 1992.

[Mon00] Joe J. Monaghan. “SPH without a Tensile Instability.” In: Journal of Computational Physics (2000).

[Mül08] Matthias Müller. “Hierarchical Position Based Dynamics.” In: VRIPHYS. Eurographics Association, 2008.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. “Particle-based Fluid Simulation for Interactive Applications.” In: Symposium on Computer Animation. 2003.

[Mül+05] Matthias Müller et al. “Particle-Based Fluid-Fluid Interaction.” In: Symposium on Computer Animation. 2005.

[Mül+07] Matthias Müller et al. “Position Based Dynamics.” In: J. Vis. Comun. Image Represent. (2007).

[Mun+09] Asim Munawar et al. “Hybrid of genetic algorithm and local search to solve MAX-SAT problem using nVidia CUDA framework.” In: Genetic Programming and Evolvable Machines (2009).

[NP94] Richard. P. Nelson and John C. B. Papaloizou. “Variable Smoothing Lengths and Energy Conservation in Smoothed Particle Hydrodynamics.” In: Monthly Notices of the Royal Astronomical Society (1994).

[Ngu07] Hubert Nguyen. GPU Gems 3. Addison-Wesley Professional, 2007.

[NNH10] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. Principles of Program Analysis. Springer Publishing Company, Incorporated, 2010.

[NQC15] Clara Novoa, Apan Qasem, and Abhilash Chaparala. “A SIMD Tabu Search Implementation for Solving the Quadratic Assignment Problem with GPU Acceleration.” In: Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure. 2015.

[NVI14] NVIDIA. Faster Parallel Reductions on Kepler. 2014.

[NVI23a] NVIDIA. CUDA C Programming Guide v11.8. 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 05/06/2023).

[NVI23b] NVIDIA. NVIDIA Ampere Architecture. 2023. URL: https://www.nvidia.com/en-us/data-center/ampere-architecture/ (visited on 05/06/2023).

[NVI] NVIDIA. Write Flexible Kernels with Grid-Stride Loops. URL: https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/ (visited on 11/22/2021).

[ONI05] Shigeru Owada, Frank Nielsen, and Takeo Igarashi. “Volume Catcher.” In: Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games. 2005.

[Pie02] Benjamin C. Pierce. Types and Programming Languages. The MIT Press, 2002.

[Pie+97] Jeffrey S. Pierce et al. “Image Plane Interaction Techniques in 3D Immersive Environments.” In: Proceedings of the 1997 Symposium on Interactive 3D Graphics. 1997.

[POC05] Fábio Policarpo, Manuel M. Oliveira, and João L. D. Comba. “Real-time relief mapping on arbitrary polygonal surfaces.” In: Proceedings of the 2005 symposium on Interactive 3D graphics and games. ACM, 2005.

[PB15] Justin Pounders and Joseph Boffie. “Analysis Of An Adaptive Time Step Scheme For the Transient Diffusion Equation.” In: 2015.

[PCW17] Edward Powley, Peter Cowling, and Daniel Whitehouse. Memory Bounded Monte Carlo Tree Search. 2017.

[Pre+03] Simon Premoze et al. “Particle-Based Simulation of Fluids.” In: Proceedings of Eurographics, Computer Graphics Forum. 2003.

[RT17] Mohammad Harun Rashid and Lixin Tao. “Parallel Combinatorial Optimization Heuristics with GPUs.” In: International Symposium on Computer Science and Intelligent Controls (ISCSIC). 2017.

[RT18] Mohammad Harun Rashid and Lixin Tao. “Parallel Combinatorial Optimization Heuristics with GPUs.” In: Advances in Science, Technology and Engineering Systems Journal (2018).

[RE13] Minsoo Rhu and Mattan Erez. “Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation.” In: (2013).

[Rob+08] Avi Robinson-Mosher et al. “Two-way Coupling of Fluids to Rigid and Deformable Solids and Shells.” In: ACM Transactions on Graphics (2008).

[RS10] Kamil Rocki and Reiji Suda. “Massively Parallel Monte Carlo Tree Search.” In: Proceedings of the 9th International Meeting High Performance Computing for Computational Science (2010).

[Sil+16] David Silver et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search.” In: Nature (2016).

[SG11] Barbara Solenthaler and Markus Gross. “Two-Scale Particle Simulation.” In: ACM Siggraph. 2011.

[SP08] Barbara Solenthaler and Renato Pajarola. “Density Contrast SPH Interfaces.” In: Eurographics/SIGGRAPH Symposium on Computer Animation. 2008.

[SP09] Barbara Solenthaler and Renato Pajarola. “Predictive-Corrective Incompressible SPH.” In: ACM Siggraph. 2009.

[SP04] Anthony Steed and Chris Parker. “3D Selection Strategies for Head Tracked and Non-Head Tracked Operation of Spatially Immersive Displays.” In: 8th International Immersive Projection Technology Workshop. 2004.

[Tal09] El-Ghazali Talbi. Metaheuristics: From Design to Implementation. Wiley Publishing, 2009.

[TCS20] Anita Tino, Caroline Collange, and André Seznec. “SIMT-X: Extending Single-Instruction Multi-Threading to Out-of-Order Cores.” In: ACM Transactions on Architecture and Code Optimization 17 (2020).

[Tuk77] John W. Tukey. Exploratory Data Analysis. 1977.

[Val+10] Dimitar Valkov et al. “Touching Floating Objects in Projection-based Virtual Reality Environments.” In: Joint Virtual Reality Conference of EGVE - EuroVR - VEC. 2010.

[VMM99] J. Vollmer, R. Mencl, and H. Müller. “Improved Laplacian smoothing of noisy surface meshes.” In: Computer Graphics Forum. 1999.

[Wal11] Ingo Wald. “Active Thread Compaction for GPU Path Tracing.” In: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics. Association for Computing Machinery, 2011.

[WVH11] Alexander Wiebel, Frans M. Vos, and Hans-Christian Hege. Perception-Oriented Picking of Structures in Direct Volumetric Renderings. Tech. rep. ZIB, 2011.

[WHK16] R Winchenbach, H. Hochstetter, and A. Kolb. “Constrained Neighbor Lists for SPH-based Fluid Simulations.” In: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation. 2016.

[XMM18] Chenjun Xiao, Jincheng Mei, and Martin Müller. Memory-Augmented Monte Carlo Tree Search. 2018.

[Yu+12] Lingyun Yu et al. “Efficient Structure-Aware Selection Techniques for 3D Point Cloud Visualizations with 2DOF Input.” In: IEEE Transactions on Visualization and Computer Graphics (2012).

[Yu+16] Lingyun Yu et al. “CAST: Effective and Efficient User Interaction for Context-Aware Selection in 3D Particle Clouds.” In: IEEE Transactions on Visualization and Computer Graphics (2016).

[ZSP08] Yanci Zhang, Barbara Solenthaler, and Renato Pajarola. “Adaptive Sampling and Rendering of Fluids on the GPU.” In: Proceedings of Eurographics / IEEE VGTC Conference on Point-Based Graphics. 2008.

[ZZ15] Yichao Zhou and Jianyang Zeng. Massively Parallel A* Search on a GPU. 2015.