GPU Data Structures and Code Generation for Modeling, Simulation, and Visualization

Thesis · February 2020

DOI: 10.25534/tuprints-00011291

1 author:

Johannes Sebastian Mueller-Roemer
Fraunhofer Institute for Computer Graphics Research IGD
18 PUBLICATIONS **60** CITATIONS

GPU Data Structures and Code Generation for Modeling, Simulation, and Visualization

GPU-Datenstrukturen und Code-Generierung für Modellierung, Simulation und Visualisierung
Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Johannes S. Mueller-Roemer aus Wilmington, NC, USA
20. Dezember 2019 — Darmstadt — D17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

GPU Data Structures and Code Generation for Modeling, Simulation, and Visualization
GPU-Datenstrukturen und Code-Generierung für Modellierung, Simulation und Visualisierung

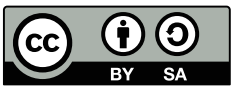
Genehmigte Dissertation von Johannes S. Mueller-Roemer aus Wilmington, NC, USA

1. Gutachten: Prof. Dr. techn. Dr.-Ing. eh. Dieter W. Fellner
2. Gutachten: Hon. Prof. Dr.-Ing. André Stork
3. Gutachten: Prof. Dr. Heinrich Müller

Tag der Einreichung: 4.11.2019
Tag der Prüfung: 16.12.2019

Darmstadt — D17

Jahr der Veröffentlichung der Dissertation auf TUprints: 2020
URN: urn:nbn:de:tuda-tuprints-112918



Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 und § 23 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Johannes S. Mueller-Roemer, die vorliegende Dissertation gemäß § 22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß § 23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Datum: Unterschrift:

_____ _____

Abstract

Virtual prototyping, the iterative process of using computer-aided (CAx) modeling, simulation, and visualization tools to optimize prototypes and products before manufacturing the first physical artifact, plays an increasingly important role in the modern product development process. Especially due to the availability of affordable additive manufacturing (AM) methods (3D printing), it is becoming increasingly possible to manufacture customized products or even for customers to print items for themselves. In such cases, the first physical prototype is frequently the final product.

In this dissertation, methods to efficiently parallelize modeling, simulation, and visualization operations are examined with the goal of reducing iteration times in the virtual prototyping cycle, while simultaneously improving the availability of the necessary CAx tools. The presented methods focus on parallelization on programmable graphics processing units (GPUs). Modern GPUs are fully programmable massively parallel manycore processors that are characterized by their high energy efficiency and good price-performance ratio. Additionally, GPUs are already present in many workstations and home computers due to their use in computer-aided design (CAD) and computer games. However, specialized algorithms and data structures are required to make efficient use of the processing power of GPUs.

Using the novel GPU-optimized data structures and algorithms as well as the new applications of compiler technology introduced in this dissertation, speedups between approximately one ($10\times$) and more than two orders of magnitude ($> 100\times$) are achieved compared to the state of the art in the three core areas of virtual prototyping. Additionally, memory use and required bandwidths are reduced by up to nearly 86%. As a result, not only can computations on existing models be executed more efficiently but larger models can be created and processed as well.

In the area of modeling, efficient discrete mesh processing algorithms are examined with a focus on volumetric meshes. In the field of simulation, the assembly of the large sparse system matrices resulting from the finite element method (FEM) and the simulation of fluid dynamics are accelerated. As sparse matrices form the foundation of the presented approaches to mesh processing and simulation, GPU-optimized sparse matrix data structures and hardware- and domain-specific automatic tuning of these data structures are developed and examined as well. In the area of visualization, visualization latencies in remote visualization of cloud-based simulations are reduced by using an optimizing query compiler. By using hybrid visualization, various user interactions can be performed without network round trip latencies.

Zusammenfassung

Virtual Prototyping, der iterative Prozess der rechnergestützten (englisch: computer-aided (CAx)) Modellierung, Simulation und Visualisierung, um Prototypen und Produkte vor der ersten Fertigung zu optimieren, nimmt im modernen Produktentwicklungsprozess eine immer größere Rolle ein. Insbesondere aufgrund der Verfügbarkeit von preisgünstigen additiven Fertigungsverfahren (3D-Druck) wird es zunehmend möglich, Produkte kundenspezifisch herzustellen oder sogar als Endkunde selber zu drucken. In solchen Fällen ist der erste reale Prototyp oftmals das Endprodukt.

Um die iterativen Zyklen des Virtual Prototyping zu verkürzen, wird in dieser Dissertation untersucht, wie Verfahren aus den Bereichen Modellierung, Simulation und Visualisierung mittels Parallelisierung effizienter durchgeführt werden können. Gleichzeitig soll dabei die Verfügbarkeit der dazu notwendigen rechnergestützten Werkzeuge verbessert werden. Der Fokus liegt dabei auf der Parallelisierung auf programmierbaren Graphikprozessoren (englisch: graphics processing units (GPUs)). Moderne Graphikprozessoren sind voll programmierbare massiv-parallele Manycore-Prozessoren, die sich sowohl durch ihre hohe Energieeffizienz als auch ihr gutes Preis-Leistungs-Verhältnis auszeichnen. Zudem sind sie durch ihre Verwendung in rechnergestützten Konstruktionsprogrammen (englisch: computer-aided design (CAD) software) sowie Computerspielen bereits in vielen Workstations und privaten Rechnern vorhanden. Jedoch sind spezialisierte Algorithmen und Datenstrukturen notwendig, um die hohe Rechenleistung von GPUs effizient auszunutzen.

Durch die Verwendung der in dieser Dissertation vorgestellten neuartigen GPU-optimierten Datenstrukturen und Algorithmen sowie neuen Anwendungen der Compilertechnik werden in den drei Kernbereichen des Virtual Prototyping Beschleunigungen zwischen etwa einer ($10\times$) und mehr als zwei Größenordnungen ($> 100\times$) im Vergleich zum Stand der Technik erreicht. Auch Speicherverbrauch und Übertragungsbandbreiten werden um bis zu knapp 86% reduziert. Somit kann nicht nur mit existierenden Modellen schneller gerechnet werden, sondern es können auch größere Modelle erzeugt und verarbeitet werden.

Im Bereich der Modellierung wird die effiziente parallele Verarbeitung von diskreten und insbesondere volumetrischen Netzen untersucht. In der Simulation werden die Aufstellung der großen dünnbesetzten Systemmatrizen aus der Finite-Elemente-Methode (FEM) und die Simulation von Fluiddynamik beschleunigt. Da dünnbesetzte Matrizen die Grundlage für die vorgestellten Ansätze zur Netzverarbeitung und der Simulation bilden, werden auch GPU-optimierte Datenstrukturen für dünnbesetzte Matrizen sowie das automatische Abstimmen solcher Datenstrukturen auf die vorhandene Hardware und das domänenspezifische Anwendungsfeld entwickelt und untersucht. Im Bereich der Visualisierung wird die Latenz bei der entfernten Visualisierung von Cloud-basierten Simulationen mittels eines optimierenden Anfragencompilers reduziert. Durch hybride Visualisierung können verschiedene Benutzerinteraktionen ohne Netzwerklatenz durchgeführt werden.

Contents

1. Introduction

Modeling, simulation, and visualization, the core components of the virtual prototyping cycle shown in Fig. 1.1, continue to become more and more important in both engineering and the creation of computer generated imagery for movies and television. Especially with the widespread availability of affordable additive manufacturing (AM) methods (3D printing), one-off customized products, small production runs, and topologically optimized parts are quickly becoming more common (cf. [ALM+17; Kra17; OMGF18]). For customized items and small runs with mechanical requirements, however, destructive testing methods, often used in conventional, large volume manufacturing, are no longer a viable approach. Furthermore, topologically optimizing a part requires running hundreds or even thousands of simulations. Combined with the fact that 3D printing is affordable and accessible to everyone, we will require more and faster modeling, simulation, and visualization tools in the future and they will have to be more readily available to a broad spectrum of users.

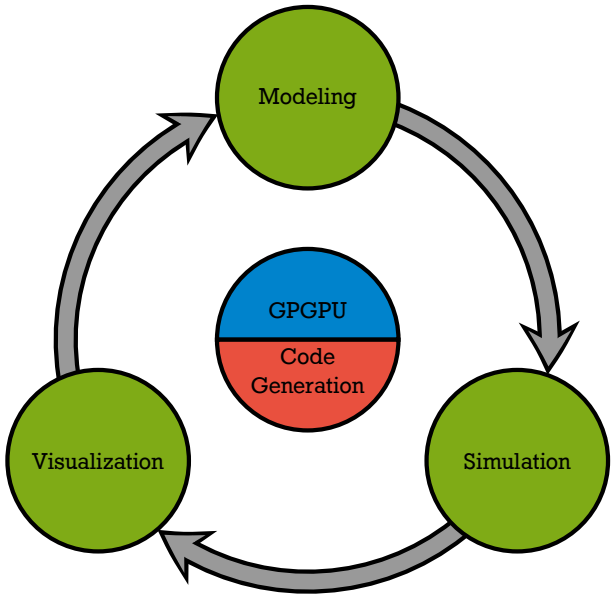


Figure 1.1.: Schematic representation of the virtual prototyping cycle and the approaches examined in this thesis to accelerate it.

There are several definitions of the meaning of virtual prototyping (see, e.g., [ZWPG03]). Here, we use the following definition:

> Virtual Prototyping is a "software-based engineering discipline that entails modeling a mechanical system, simulating and visualizing its 3D motion behavior under real-world operating conditions, and refining and optimizing the design through iterative design studies before building the first physical prototype." [LaC01]

In the context of AM, virtual prototyping takes on an even greater role, as the first physical prototype may be the final product.

As the goal of this thesis is to improve both speed and availability of virtual prototyping tools, especially for customization and small production runs, the focus of this thesis lies on small to medium-sized meshes and commodity hardware present in desktop computers and workstations. While the focus of research in the high performance computing (HPC) community is shifting to alternative co-processor architectures such as field programmable gate arrays (FPGAs), graphics processing units (GPUs) and other manycore architectures (e.g., PEZY SC-2, Intel Xeon Phi) continue to dominate the top 25 entries of the Green500 list [FS19]. The Green500 list corresponds to the Top500 list [SDS+19], a list of the 500 fastest commercially available computer systems, sorted by floating point operations per second (FLOPS) per watt. Compared to other manycore architectures, GPUs are more readily available and significantly cheaper when single

precision floating point is sufficient, as low cost consumer-level GPUs that are commonly used for gaming can be employed.

Due to their availability, performance per watt, and performance per dollar, GPUs make an ideal choice to accelerate the virtual prototyping cycle. Furthermore, many scientific visualization methods map naturally to graphics hardware and application programming interfaces (APIs). Additionally, GPUs are typically already present in computer-aided design (CAD) workstations and consumer gaming systems. But many steps are still performed, often serially, on the system processor (CPU). However, specialized algorithms and data structures are required to efficiently use GPUs due to their performance characteristics (see Section 2.1). Furthermore, these characteristics can vary significantly depending on the specific GPU on hand, often requiring just-in-time (JIT) code generation to achieve optimal performance.

This leads to the main research question of this doctoral thesis:

> **Can the available hardware, particularly manycore GPUs, be used more efficiently in the virtual prototyping cycle?** If yes, which components of the virtual prototyping cycle can be improved in what manner?

As this question is very broad, we must break it down further. Additionally, each of the steps of the virtual prototyping cycle covers a vast range of research topics itself. Therefore, we must first analyze the individual steps to determine which parts of them could benefit the most from (improved) GPU parallelization, especially those for which there are no GPU implementations and are necessary to "close the loop," and which ones are out of the scope of this dissertation.

In conventional computer-aided (CAx) product development workflows, an object is first modeled as a boundary representation (B-rep) in a CAD tool. Interactive modeling of B-reps only affects a small number of entities and is not primarily limited by performance, but by the user and user interface design, as well as the numerical robustness of the operations (see, e.g., [UMC+19]). However, geometry processing of discrete meshes, as used in modelers based on triangle meshes such as Meshmixer [Aut18], require processing many entities at once. Efficient mesh processing on the GPU is challenging due to the sparseness and irregularity of element neighborhood relationships.

After modeling, product models are imported into a computer-aided engineering (CAE) tool. CAE tools use various numerical methods, e.g., the finite element method (FEM), to simulate physical properties such as deformation, heat transfer, or fluid dynamics. These methods operate on discrete and often volumetric meshes, further highlighting the importance of efficient GPU mesh data structures. When working with B-reps, import involves meshing the model, i.e., converting it into a discrete, volumetric mesh. GPU-accelerated meshing has been explored by other authors [Nan12; CNGT14], but meshing often suffers from robustness issues that require manual modifications to the input, negating any performance gains (see, e.g., [HZG+18]). Alternatively, it is possible to model directly in the volumetric domain (see, e.g., [ASSF17]) or to apply mesh morphing to create prototype variations (see, e.g., [MO07]).

After meshing (or volumetric modeling), the user specifies boundary conditions and physical properties. This process is called pre-processing in CAE. The simulation process itself and the required solution of large, sparse systems of equations have been the focus of extensive research in HPC and general pur-

pose computing on the GPU (GPGPU). In fact, simulation was one of the first applications of the GPGPU programming language CUDA [LJWD08]. However, the aforementioned systems of equations must first be assembled, typically in the form of a sparse matrix or tensor. Besides the sparseness and irregularity inherited from unstructured meshes, CPU assembly methods typically require dynamic allocation during execution, which does not map well to the GPU execution model. Existing state-of-the-art methods involve significant overallocation to avoid dynamic allocation, creating a large memory overhead and limiting maximum simulation resolution [ZSS17a; ZSS17b].

Efficient GPU-optimized sparse matrix data structures are relevant to geometry processing methods used in modeling and to the entire simulation step, both during assembly and system solution. Due to the ubiquity of sparse matrices and the variance in GPU performance characteristics, it is worth exploring how code generation can be used to create more efficient data structures and code for the specific GPU used. Furthermore, simulation system matrices often exhibit local structure with dense blocks and can additionally require extended number systems, i.e., complex numbers or quaternions. Both of these can be exploited in specialized data structures to further improve performance and decrease memory use.

After simulation, the results must be evaluated by visualizing or analyzing them using other post-processing methods. Most scientific visualization methods map naturally to graphics hardware and APIs. Therefore, current research focuses either on out-of-core methods (see, e.g., [SCRL19]) for very large datasets more relevant to detail simulations than prototyping or interaction and perception (see, e.g., [SKR18]), which is out of the scope of this dissertation. However, the computation of derived values, which are often domain and use case specific and therefore not always known a priori, can create significant bandwidth overhead when performed on the CPU. This bandwidth overhead becomes even more significant for simulations running at interactive rates and when using remote visualization. Remote visualization is becoming increasingly relevant due to simulations running in the cloud (servers leased on demand) that increase availability to individuals and small companies due to reduced HPC operating costs. Based on the evaluation, the original model is iteratively modified as necessary until a satisfactory result is achieved.

Therefore we break the main research question down into the following sub-questions:

1. **Can the GPU be used to efficiently process unstructured meshes, both polyhedral meshes in general and tetrahedral meshes in particular?** If yes, which mesh data structures and algorithms are suitable for GPU processing?

2. **Can these GPU-optimized data structures be used to perform system matrix assembly for the FEM and other simulation methods more efficiently?** If yes, how can memory overhead be reduced while maintaining or improving performance?

3. **Can code generation and compiler techniques be used to efficiently implement GPU sparse matrix formats and algorithms required in simulation and mesh processing?** Specifically, how can the performance of sparse matrices with extended number systems (e.g., complex numbers and quaternions) and dense blocks be improved?

4. **Can GPGPU and code generation for the GPU be used to improve the performance of remote post-processing and visualization?** In particular, how can bandwidth overheads be minimized and GPU performance be exploited when user queries are only known at runtime?

The remainder of the introduction lists contributions and the publications in which they were first presented in Section 1.1, and provides an outline of this dissertation's structure in Section 1.2.

1.1. Contributions

This doctoral thesis contains contributions to each of the core components of the virtual prototyping cycle, as detailed in Sections 1.1.1 to 1.1.3. The approaches used to achieve these contributions fall into the categories of either data structures and algorithms for GPGPU or code generation, as indicated in Fig. 1.1. As efficient GPU-accelerated sparse matrix data structures are relevant to both modeling and simulation, the contributions to that field are detailed separately in Section 1.1.4. Finally, relevant publications are listed in Section 1.1.5.

1.1.1. Modeling

Conventional, interactive modeling in CAD tools, e.g., the addition, modification, or removal of individual faces or patches, cannot benefit significantly from parallelization due to the small number of entities affected. However, modeling tools based on triangle meshes (see, e.g., [SB16; Aut18]) or volumetric meshes (see, e.g., [ALM+17]) frequently process many entities of the mesh at once. In the area of modeling, specifically volumetric mesh processing, the contributions are:

1. A novel data structure for parallel processing of general polyhedral meshes on the GPU based on a compact encoding of boundary operator matrices is introduced in Chapter 3. Compared to the state of the art in array-based data structures for general polyhedral meshes, memory use is reduced by up to 36% and speedups of up to 531× are achieved for neighborhood queries. The improvements also carry over to parallel implementations on the CPU, achieving speedups of up to 149×.
2. Using the novel data structure, several parallel mesh processing algorithms have been implemented.
   a) Laplacian smoothing of inner vertices, as used to improve element quality after deformation in mesh morphing approaches, achieves a speedup of up to 289× (48× on the CPU).
   b) Boundary surface extraction, which is used to efficiently visualize simulation meshes, is sped up by a factor of up to 8× (5× on the CPU).
   c) Volumetric Catmull-Clark subdivision, usable in multiresolution modeling and required when increasing mesh resolution for subdivision-based simulation, is up to 166× faster (59× on the CPU).

1.1.2. Simulation

While GPGPU has been applied to simulation early on (see, e.g., [LJWD08]), the assembly process of the system matrix itself has only recently come into the focus of research (see, e.g., [GLG+15]). Assembling the system matrix involves determining the sparsity pattern and summation of element matrices. Especially the determination of the sparsity pattern is typically still performed serially on the CPU. With respect to simulation and system matrix assembly, the contributions are:

1. An approach to calculating the number of non-zero entries per row exactly using minimal topological information for simplex meshes of arbitrary polynomial degree is presented in Chapter 4. This

makes it possible to determine the sparsity pattern in parallel, without the up to 600% memory overhead of the current state-of-the-art approach to GPU-based system matrix assembly [ZSS17b]. Furthermore, it enables direct assembly into GPU-optimized sparse matrix data structures while simplifying implementation.

2. Combined with a specialized variant for simplex meshes of the ternary matrix mesh data structure described in Section 3.4.4, the exact allocation approach to matrix assembly is shown to provide a significant speedup compared to the current state of the art in matrix assembly. Compared to serial CPU-based assembly, speedups of up to 200× are achieved.

3. Three different summation approaches are compared in Section 4.5.2, providing clear guidelines for the choice of summation method, depending on the type (static or dynamic) and polynomial degree of the simulation.

4. An improved finite volume method (FVM) formulation for fluid simulation with cut cells is presented in Section 6.3.2. When used in a geometric multigrid solver, the discretization leads to a consistent multigrid hierarchy and high convergence rates. The resulting solver has been shown to be up to 3× faster than the state of the art [WMSF15; Web16].

1.1.3. Visualization

Besides the efficient use of readily available commodity hardware, a complementary approach to improve access to high-performance virtual prototyping tools is remote visualization of simulations running in the cloud. In such setups, the user only requires a low-cost client machine, potentially a mobile phone, running a browser or native client software for better performance. The contributions in the field of remote visualization of GPU-accelerated simulations running at interactive rates are:

1. An optimizing query compiler for remote visualization of derived fields is presented in Chapter 6. Similar approaches have been used in visual analytics [MŞ15] and more recently in geospatial data visualization [LGMF17], but not in the field of remote scientific visualization. Compared to using an interpreter, execution times and therefore latency are reduced by a factor of 14×. Compared to compiling queries to CPU code, calculation are accelerated by up to 20×, but the main gains come from reducing data transfer costs from GPU to CPU which account for up to 72.3% of runtime in the evaluation.

2. Prototypical streaming clients implemented in both native C++ and browser-based HyperText Markup Language 5 (HTML5) variants are evaluated in Chapter 6. Combined with the efficient GPU-based multigrid fluid solver discussed in Section 6.3.2, interactive visualization frame rates (≥ 10 frames per second) are achieved. Through the use of a hybrid rendering approach, many interactions can be performed with zero latency.

3. An optimized encoding for hybrid rendering of 3D simulation data is presented in Section 6.3.5. The presented method extends the rich pixel (rixel) streaming approach by Altenhofen et al. [ADSF16]. Compared to the original encoding, message size is reduced by up to 59% (73% if 16-bit depths are sufficient). By performing encoding on the GPU, bandwidth savings of *at least* 59% apply to GPU to CPU copies as well.

1.1.4. Sparse Matrix Data Structures

Efficient sparse matrix data structures and operations form the foundation of both simulation and the novel sparse ternary matrix-based mesh representation for volumetric meshes presented in this thesis. The contributions in the area of (GPU-optimized) sparse matrix data structures are:

1. A compact encoding of ternary sparse matrices, ternary compressed sparse row (TCSR), that is efficient to decode and forms the basis of the mesh data structure presented in Chapter 3.
2. An improved version of the Bin-BCSR data structure for matrices with dense blocks by Weber et al. [WBS+13], Bin-BCSR*, is presented in Chapter 4. Bin-BCSR* is more compact than Bin-BCSR, improves locality, and enables dynamic scheduling, which can lead to additional performance benefits on highly irregular matrices.
3. A code generator for sparse matrix data structures with compound entries is introduced in Chapter 5. The code generator enables joint schedule and layout tuning, leading to speedups of up to 4.7× compared to the highly tuned vendor library. Compared to schedule tuning without layout tuning, a speedup of up to 5.5× is achieved.

1.1.5. Publications

Chapters 3 to 6 of this dissertation are based on a number of works previously published in journals or conference proceedings:

[WMSF15] Weber, D., **J. S. Mueller-Roemer**[1], A. Stork, and D. W. Fellner.
"A Cut-Cell Geometric Multigrid Poisson Solver for Fluid Simulation."
In: *Computer Graphics Forum* 34(2) (Eurographics 2015), pp. 481–491.
DOI: 10.1111/cgf.12577.

[MA16] **Mueller-Roemer, J. S.** and C. Altenhofen.
"JIT-compilation for Interactive Scientific Visualization."
In: *Short Papers Proceedings: 24th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*. WSCG '16. 2016, pp. 197–206.

[MAS17] **Mueller-Roemer, J. S.**, C. Altenhofen, and A. Stork. "Ternary Sparse Matrix Representation for Volumetric Mesh Subdivision and Processing on GPUs." In: *Computer Graphics Forum* 36(5) (Symposium on Geometry Processing 2017), pp. 59–69.
DOI: 10.1111/cgf.13245.

[MS18] **Mueller-Roemer, J. S.** and A. Stork.
"GPU-based Polynomial Finite Element Matrix Assembly for Simplex Meshes."
In: *Computer Graphics Forum* 37(7) (Pacific Graphics 2018), pp. 443–454.
DOI: 10.1111/cgf.13581.

[BGM19] Bormann, P., R. Gutbell, and **J. S. Mueller-Roemer**.
"Integrating Server-based Simulations into Web-based Geo-applications."
In: *Eurographics 2019 - Short Papers*. 2019. DOI: 10.2312/egs.20191012.

[1] The two primary authors contributed equally to this work.

[MSF19] **Mueller-Roemer, J. S.**, A. Stork, and D. W. Fellner. "Joint Schedule and Layout Autotuning for Sparse Matrices with Compound Entries on GPUs."
In: *Vision, Modeling and Visualization*. VMV '19. 2019, pp. 109–116.
DOI: 10.2312/vmv.20191324.

Large portions of these publications are quoted verbatim, including both text and illustrations, with minor changes, additions, and corrections, as noted in the individual chapter introductions. A complete list of all publications I have (co-)authored and conference talks I have held at the time of writing this dissertation can be found in Appendix B.

1.2. Structure

The structure of this doctoral thesis closely follows the virtual prototyping cycle of modeling, simulation, and visualization. In the following, Chapter 2 provides a background on GPU architecture and GPGPU programming models and sparse matrix data structures. Chapter 3 discusses related work in the areas of mesh data structures and processing on GPUs and provides a background on volumetric subdivision in Section 3.2. In the following sections, Chapter 3 describes a novel data structure for GPU-parallelized processing of general polyhedral meshes that is based on sparse matrices and that is applicable to volumetric modeling and mesh processing. After outlining related work in the areas of GPU-optimized and system matrix assembly, as well as providing a background on the polynomial FEM, Chapter 4 describes a highly efficient GPU-based sparse matrix assembly method for mesh-based simulations that avoids excess allocation of memory by exploiting the topological properties of simplicial meshes. Furthermore, enhancements to the Bin-BCSR data structure are described in Section 4.4.2. After discussing related work on layout autotuning and code generation for sparse matrix operations, Chapter 5 presents a code generation technique for automatic tuning of sparse matrix data structures with compound entries, as occur both in geometric processing and simulations based on the FEM. Chapter 6 outlines related work and alternative approaches in the areas of compiler technologies for visualization, floating point data compression, and application sharing in Section 6.2. Furthermore, it covers how code generation and optimizing compiler technologies can be applied to improve remote visualization of interactive simulation methods. The improved FVM discretization for fluids with cut cells used in the simulation backend is described in Section 6.3.2. Finally, Chapter 7 concludes the thesis and highlights avenues for further research in all core areas of the virtual prototyping cycle in Section 7.1.

2. Background

This chapter provides a background on graphics processing unit (GPU) architecture and general purpose computing on the GPU (GPGPU) programming models in Section 2.1 as well as sparse matrix data structures in Section 2.2. The former is relevant to all following chapters, while the latter is relevant to Chapters 3 to 5. Background and related work specific to the individual chapters is provided in Sections 3.2, 4.2, 5.2, and 6.2.

2.1. GPU Architecture and Programming Model

Compared to multicore system processors (CPUs) which offer a small number of powerful cores, i.e., individual processors on a single die, manycore processors have a large number of significantly simpler and slower cores and rely on parallelism to achieve a high throughput at the cost of increased latency. And while CPU core counts and single instruction multiple data (SIMD) vector widths are steadily increasing, and manycore processors are increasing in complexity (and bootable manycore CPUs such as Intel's Xeon Phi Knight's Landing (KNL) being available [Sod15], albeit discontinued [Mor18]), the architectures and more importantly the programming models remain quite disparate.

In the following, Section 2.1.1 describes the hardware architecture of GPUs. Section 2.1.2 lists available programming models and languages and describes the CUDA language and programming model in more detail. Section 2.1.3 collects important performance considerations that result from the architecture and programming model for ease of reference.

2.1.1. GPU Architecture

GPUs are specialized manycore co-processors, i.e., processors used in addition to a conventional CPU, designed to accelerate 3D-graphics applications, such as games and computer-aided design (CAD) software. As such, they include dedicated circuitry for triangle rasterization, texture fetching and interpolation, and other graphics-oriented operations such as tessellation (see, e.g., [NVI16]). At the same time, modern GPUs have evolved to also be highly parallel, programmable, general purpose processors with high computational throughput and memory bandwidth.

The following description of GPU architecture is based on the description and information given in the CUDA C Programming Guide [NVI18a], as all data structures and algorithms described in this thesis were implemented in CUDA and evaluated on NVIDIA GPUs. However, other GPU architectures share many similarities (compare, for example, AMD's Vega10 architecture [MS17] and NVIDIA's Volta architecture [Cho17]), especially at the level of abstraction of this description. Therefore, there should be no loss of generality. A detailed comparison of two older AMD and NVIDIA GPU architectures has been performed by Zhang et al. [ZPL+11].

GPUs achieve their high computational throughput by dedicating a significantly larger percentage of their die area to arithmetic logic units (ALUs), especially floating point units, than to control flow and caching,

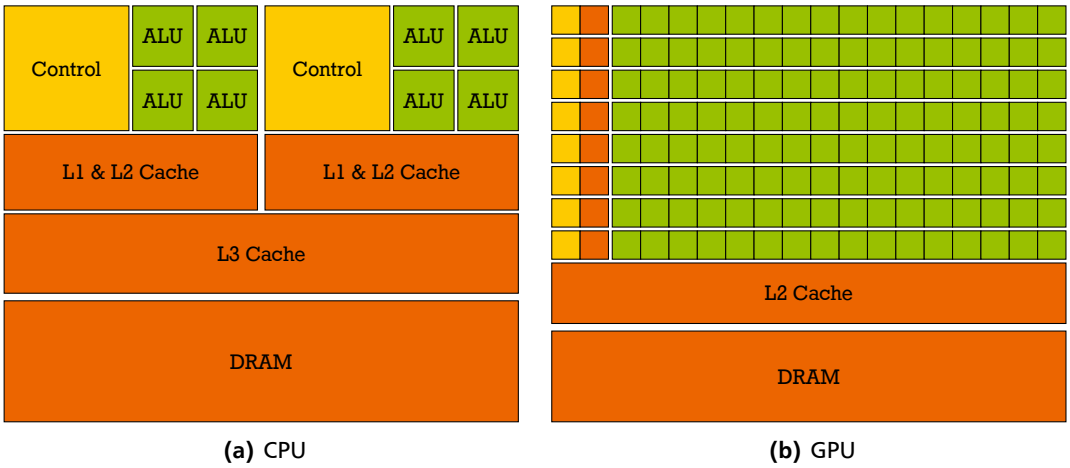


Figure 2.1.: Simplified comparison of **(a)** multicore CPU and **(b)** manycore GPU architectures based on Fig. 3 of the CUDA C Programming Guide [NVI18a], adding top level caches and a second CPU core. GPUs devote a larger percentage of their die area to arithmetic, especially floating point, units.

as illustrated in Fig. 2.1. Due to this design, GPUs are particularly well-suited for data-parallel programs with high arithmetic intensity, i.e., a high ratio of arithmetic operations to bytes of data read or written, and simple control flow. Unlike CPUs, instructions are executed in order and neither branch prediction nor speculative execution are performed. The high memory bandwidth is achieved by using a very wide memory bus. For example, the NVIDIA GP100 [NVI17] and GV100 [NVI18c] GPUs each have eight 512-bit memory controllers.

The NVIDIA GPU architecture is organized around a scalable array of streaming multiprocessors (SMs). Each SM is designed to execute hundreds of threads in parallel using a single instruction multiple thread (SIMT) architecture. In this architecture, the SMs create, manage, schedule, and execute threads in groups of 32 parallel threads called warps. While each warp has its own register state, each warp shares a single instruction counter and executes one common instruction at a time. With the Volta architecture, per-thread instruction counters have been introduced. However, each warp still executes one common instruction at a time. Therefore, if threads of a warp diverge due to branching control flow, the warp executes each branch path taken, disabling (masking) threads that are not on the current path. Separate warps execute independently, regardless of any diverging control flow.

Therefore, SIMT is very similar to SIMD architectures with masking, such as Intel's AVX-512 instruction set [Int18]. The main difference is that SIMD instruction sets explicitly encode the vector width, while SIMT instructions specify the behavior of a single thread. Therefore, for the purposes of correctness, it is generally not necessary to know the size of a warp. However, if high performance is key, warp size and divergence, like cache line size on CPUs, must be taken into account.

Another important difference compared to CPU threads is that scheduling is performed in hardware. The execution context of each warp processed by a multiprocessor is maintained on-chip during the entire lifetime of the warp. This allows the GPU to perform execution context switching at no runtime cost. To achieve this, the multiprocessor's register file is partitioned among the warps, while cache and shared memory (essentially an explicitly programmable cache) are partitioned among the blocks (see Section 2.1.2).

To support a large number of resident threads, GPUs have very large register files of up to 512 KiB (128 Ki 32-bit registers). The closest analog for CPUs is simultaneous multithreading (SMT), such as Intel's HyperThreading [Int18], in which each physical core offers multiple logical cores by having a register file for each logical core and performing scheduling between logical cores in hardware.

The physical memory hierarchy of a GPU consists of the (non-addressable) register files, a fast per-SM L1 cache and shared memory, an L2 cache shared between the SMs, and one or more dynamic random-access memory (DRAM) chips. Current CPUs typically have one additional level of cache per core (see, e.g., [Int18]), as shown in Fig. 2.1. This physical memory hierarchy is further subdivided logically, as described in Section 2.1.2.

While the specialized hardware for rasterization and tessellation is neither relevant nor accessible when using the GPU as a general purpose manycore processor, the texturing units can be used. They support

- hardware conversion from 8- and 16-bit integers to floating point numbers in the ranges of [0,1] and [-1,1] for unsigned and signed integers, respectively,
- access using floating point addresses, potentially with hardware multilinear interpolation,
- caching optimized for access with 2D locality but without coherency (sequential access),
- and other features such as wrapping modes and limited color space conversion.

The different caching behavior is the most relevant aspect for this dissertation, as it has previously been used to accelerate the unordered access to the vector in sparse matrix-vector products (SpMVs) [BG08]. However, this altered behavior is also accessible without using texture units, which only support limited texture sizes, on modern GPUs.

2.1.2. GPGPU Programming Model

The earliest examples of GPGPU used GPUs with a fixed function graphics pipeline via graphics application programming interfaces (APIs) such as OpenGL or DirectX/Direct3D to perform other computations (see, e.g., [HKL+99]). The introduction of programmable pixel shaders in 2001 lead to a significant increase of general purpose computations possible using GPUs, but were limited by the lack of floating point support (see, e.g., [LM01]). Since the introduction of floating point color buffers in commodity hardware (see, e.g., [KW03]), several programming languages and directive-based programming models for GPGPU have been introduced.

The two main GPGPU programming languages that remain in common use are NVIDIA's CUDA [NVI18a] and Khronos' OpenCL [Khr18]. CUDA defines APIs for interaction with the GPU, extends C++ with additional keywords and syntactic constructs for GPGPU, and device (GPU) code can be defined together with the host (CPU) code to launch it. OpenCL also defines an API to launch device code and a C-based language (C++-based in OpenCL 2.2 which is not yet widely supported) with similar extensions. Unlike CUDA, the GPU code must be provided separately. However, as a vendor-neutral specification, there are implementations for non-NVIDIA GPUs, other co-processors, such as field programmable gate arrays (FPGAs), and multicore CPUs. Besides CUDA and OpenCL, major graphics APIs now offer compute shaders [SA18; Mic19], extending their respective shading languages to support GPGPU tasks. These offer a similar fea-

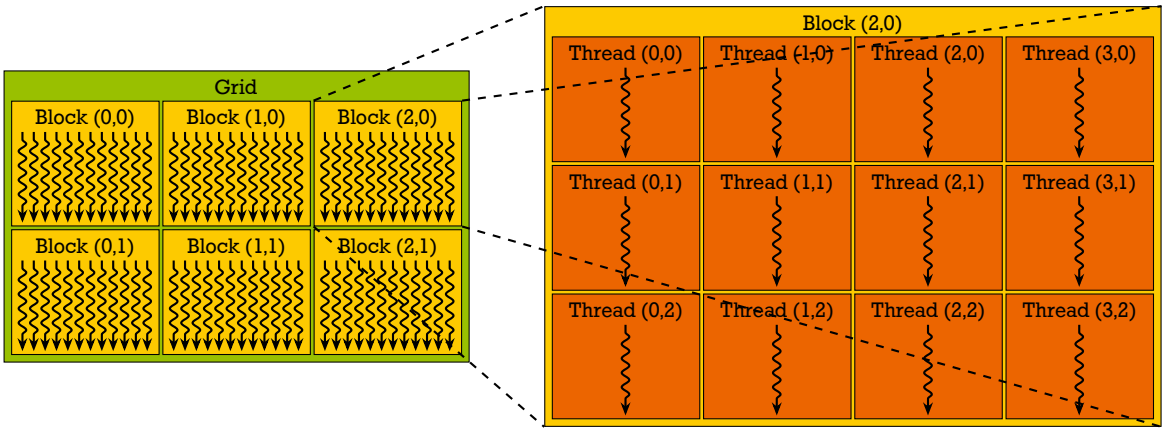


Figure 2.2.: Schematic representation of CUDA's hierarchical threading model with a 3 × 2 grid of 4 × 3 blocks. Adapted from Fig. 6 of the CUDA C Programming Guide [NVI18a].

ture set and tighter integration with the respective graphics APIs, but have not seen widespread adoption outside of graphics and games.

Directive-based models, which extend general-purpose programming languages such as C, C++, or Fortran with compiler directives that specify how the code should be parallelized, include OpenMP [Ope18b] and OpenACC [Ope18a]. OpenMP was originally designed for shared memory parallelization, but extended to support offloading to co-processors in version 4.0 [Li16]. OpenACC was designed for heterogeneous computing, but lacks support of shared memory parallelization. Directive-based models significantly reduce programmer effort, but do not currently match the performance of CUDA or OpenCL [MLP+17].

Like the description of GPU architecture in the previous section, the remainder of this section is based on the description and information given in the CUDA C Programming Guide [NVI18a], as all data structures and algorithms described in this thesis were implemented in CUDA and evaluated on NVIDIA GPUs. While OpenCL is cross-platform, it requires significantly more programmer effort. Memeti et al. have empirically determined a factor of two increase in program size [MLP+17]. Furthermore, the previous work on which this thesis builds has been implemented in CUDA as well (see, e.g., [WBS+13]). Aside from nomenclature, the CUDA and OpenCL programming models are very similar and several source translators are in development (see, e.g., [Per17]). Therefore, this choice does not limit the applicability of any of the methods presented in this thesis.

CUDA follows a single program multiple data (SPMD) parallelization model, i.e., when a GPU kernel, a function annotated with the `__global__` keyword, is launched by the host, the device runs the same function in parallel on a number of threads given by the host. Each thread is given a unique thread ID, allowing it to load a specific part of the input data. As the number of threads that can run simultaneously is limited by the number of registers available (see Section 2.1.1), CUDA uses a hierarchical threading model. Threads are organized in a one- to three-dimensional grid of equally-sized one- to three-dimensional blocks of threads, as illustrated in Fig. 2.2. On current GPUs the maximum number of threads in a block is limited to 1024 in addition to the limits implied by the required number of registers and amount of shared memory. The exact limit is determined by the compute capability (CC), which specifies which capabilities a given GPU has. Block and grid sizes are specified at runtime by the host, but must not exceed these limits.

Threads are scheduled on SMs in blocks, i.e., each SM executes zero or more blocks at any given time. Therefore, thread blocks are required to execute independently, i.e., it must be possible to execute them in any order, in parallel, or in series. This allows thread blocks to be scheduled in any order across any number of SMs. All threads within a block are resident on an SM at the same time and can access a common part of shared memory. However, before accessing shared memory written by another thread, it is necessary to synchronize the threads within the block using a `__syncthreads()` barrier. If the exchange occurs within a warp, a `__syncwarp()` memory barrier is sufficient.

Threads are assigned to warps according to their linear thread ID within a block:

$$\texttt{threadID} = \texttt{threadIdx.x} + \texttt{blockDim.x} \cdot (\texttt{threadIdx.y} + \texttt{blockDim.y} \cdot \texttt{threadIdx.z}), \tag{2.1}$$

$$\texttt{warpID} = \left\lfloor \frac{\texttt{threadID}}{32} \right\rfloor, \tag{2.2}$$

where `threadIdx` is the zero-based thread index within the block and `blockDim` is the three-dimensional block size. Besides `threadIdx` and `blockDim`, kernels have two additional implicit parameters: `blockIdx` stores the index of the current block and `gridDim` stores the size of the grid, allowing the user to compute a global thread index as well.

As mentioned in Section 2.1.1, the physical memory hierarchy is further subdivided into a number of logical memory spaces: global memory, local memory, shared memory, constant memory, and texture memory. Global memory resides in DRAM, which supports aligned 32-, 64- and 128-byte transactions, i.e., the base address of an n-byte transaction must be a multiple of n bytes. The CUDA architecture supports aligned 1-, 2-, 4-, 8-, or 16-byte load and store instructions. When a warp executes an instruction that accesses global memory, the GPU attempts coalesce the memory accesses of the threads within the warp into one or more memory transactions depending size and distribution of the memory addresses across the threads. The exact rules applied depend on the compute capability of a given GPU. Generally, when accessing a 2D array of aligned 1- to 16-byte values in column-major order ($\texttt{i} = \texttt{x} + \texttt{width} \cdot \texttt{y}$), both the width of the thread block and the width of the array must be multiples of the warp size to achieve full coalescing.

Like global memory, local memory resides in device memory, but in a reserved area partitioned by thread. Local memory is used when the indices into a local array cannot be determined at compile time, as registers are not addressable, or when register spilling occurs, i.e., when more registers would be required than are available. Generally, local memory exhibits the same high latency and low bandwidth as global memory. However, local memory is organized such that consecutive 32-bit words are accessed by consecutive thread IDs. Therefore, accesses are fully coalesced as long as all threads access the same variable or array index. However, this does not guarantee that all reads and writes are cached.

Shared memory, much like the L1 cache, resides in fast, on-chip memory. On some GPU architectures, depending on compute capability (3.x and 7.x), the hardware resources are shared between L1 cache and shared memory, and the partitioning of the two can be configured. But even in shared memory, care must be taken how accesses are performed to achieve maximum bandwidth. Shared memory is divided

into banks, and only if the n memory addresses accessed by a warp fall into n distinct banks can the full bandwidth be achieved. If two or more (distinct) addresses fall into the same bank, this is called a bank conflict and results in serialization into multiple transactions. The exact mapping between addresses and banks depends on compute capability.

Constant memory is a small 64 KiB reserved area in device memory that cannot be modified in device code. It is cached in a separate 4–8 KiB read-only constant cache. The constant cache performs best when all threads access the same address.

Texture memory also resides in device memory and is served via the texture cache. The differences in caching behavior are listed in Section 2.1.1. On GPUs of compute capability 3.5 or higher, the L1 and texture caches are unified and the `__ldg()` intrinsic allows the programmer to load immutable (during kernel execution) data from global memory through the texture cache. The compiler will also attempt to determine automatically if access is immutable via the **`const`** and **`__restrict__`** keywords. The latter informs the compiler that there can be no aliasing between that pointer and other pointers.

With respect to compilation, kernel and other device code written in CUDA is compiled to parallel thread execution (PTX) assembly and/or binary GPU code, while host code is separated out with calls to kernels replaced by a series of API calls and compiled by a regular C++ compiler. The PTX instruction set architecture (ISA) defines a low-level parallel virtual machine and corresponding instruction set, i.e., PTX assembly and registers do not map to native GPU opcodes and registers directly [NVI18e].

PTX code is JIT-compiled by the driver for the current GPU when a CUDA program run for the first time and cached on disk. The advantage of using PTX over binary code is that PTX is fully backwards compatible, while binary code is only backwards compatible within the same major compute capability version. Besides JIT-compiling OpenCL-C/C++ directly, OpenCL supports a similar approach in which SPIR-V, a binary intermediate representation for a generic parallel virtual machine, is JIT-compiled at runtime [Khr18].

To allow the CPU to perform useful work while the GPU is active, kernel launches are performed asynchronously, i.e., without blocking. Furthermore, most API methods are available in asynchronous variants. A number of synchronization primitives are offered as well.

2.1.3. GPGPU Performance Considerations

The architecture and programming model of GPUs, described in Sections 2.1.1 and 2.1.2, respectively, result in several important considerations to achieve good performance. These are collected in this section for ease of reference.

- Thread execution within a kernel is not entirely independent and occurs in warps of 32 threads. When multiple divergent code paths are taken by a warp, all active paths are executed sequentially with masking. Therefore, divergence of control flow should be avoided.
- High memory bandwidths are achieved using a wide memory bus with large transactions. To achieve good performance, data in global memory must be laid out such that threads within a warp access specific, consecutive multiples of their thread ID to achieve coalescing. In 2D arrays, this may require

adding padding to a multiple of 32 rows and transposing data from row-major order ($\texttt{i} = \texttt{y} + \texttt{height} \cdot \texttt{x}$) to column-major order ($\texttt{i} = \texttt{x} + \texttt{width} \cdot \texttt{y}$) when threads process consecutive rows.

- Memory accesses that have locality without being consecutive are best performed via texture units or non-coherent loads. However, doing so comes at the cost of increased latency and the data must be immutable for the duration of the kernel's execution.
- When loops adressing a local array cannot be unrolled, the local array is placed into local memory, a reserved segment of DRAM, as registers are not addressable. This can lead to large latencies when a read is not cached. When too many registers would be required, other variables are spilled to local memory as well.
- Shared memory is available for communication between threads within a block of threads. This low-latency, on-chip memory can also be used as an explicitly programmable cache. The organization of shared memory into banks imposes additional restrictions on memory layout when peak performance is to be achieved.
- While GPU memory bandwidths are high, the bandwidth of the peripheral component interface express (PCIe) bus connecting the GPU to the CPU is comparatively low. For example, a 16× PCIe 3.0 connection has a theoretical peak bandwidth of 16 GB/s, compared to the 900 GB/s memory bandwidth of the NVIDIA Tesla V100. Therefore, it can be beneficial to perform computations on the GPU even if it is less suitable than the CPU for a particular computation to avoid unnecessary CPU-GPU transfers.

2.2. Sparse Matrix Data Structures

Sparse matrices are matrices with a small number of non-zero entries compared to their size, i.e., matrices $\mathbf{A} \in \mathbb{F}^{n \times m}$ with $\text{nnz}(\mathbf{A}) \ll nm$, where $\mathbb{F}$ is any field and $\text{nnz}(\mathbf{A})$ is the number of non-zero entries $A_{ij} \neq 0$. Sparse matrices commonly arise in the discretization of partial differential equations (PDEs) using the finite element method (FEM) (see, e.g., [ZT00]), finite volume method (FVM) (see, e.g., [EGH00]), and other methods used in simulation. Additionally, many of these methods result in symmetric matrices. Aside from simulation, adjacency matrices and incidence matrices of graphs are sparse as well (cf. [MBB+13]). The sparsity and symmetry of these matrices can be exploited to significantly reduce both memory use and computational overhead, as zero entries do not have to be stored or processed. However, unlike dense matrices that map directly and efficiently to 2D arrays, the choice of data structure significantly influences the performance of operations on sparse matrices.

A variety of sparse matrix data structures have been in use since the 1960s and are described in the literature (see, e.g., [Saa03]). The simplest data structure or storage format for sparse matrices is the coordinate list format (COO). In this format, matrices are stored as triples of row index, column index, and value of the non-zero entries. These are stored in three separate arrays and often sorted first by row, then by column, for efficient random access by row and column index. While COO is popular as an input or exchange format due to its simplicity and is often used for matrix construction, it is ill-suited for use in further computation such as SpMVs and general sparse matrix-matrix products (SpGEMMs). One major reason is that any entry can affect any part of the output, making parallelization difficult. Furthermore, COO requires a large amount of memory, as each entry requires two indices in addition to its value.

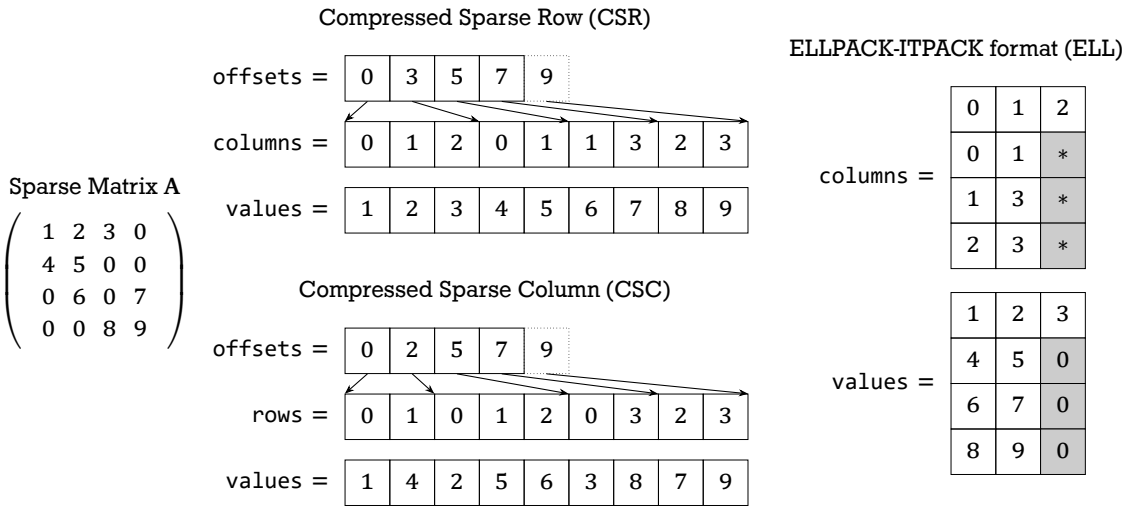


Figure 2.3.: CSR, CSC, and ELL representations of a 4 × 4 matrix $\mathbf{A}$ with nine non-zero entries and zero-based indices. The CSC representation is equivalent to the CSR representation of $\mathbf{A}^T$. The arrows indicate how the offsets relate to the starting and ending points of each row or column. Padding is shown with a gray background. Any valid column index can be used for padding. Common choices are zero and the corresponding row index.

Two of the most commonly used sparse matrix data structures for computation are the compressed sparse row (CSR) and compressed sparse column (CSC) formats. Both formats use an array of offsets into a pair of arrays: one array for the non-zero values, and one for the column- or row-indices for the CSR or CSC formats, respectively. The offsets store the starting indices of each row or column, respectively, and contain one additional entry for the end of the array. As the CSC format is identical to the CSR format aside from swapping the role of columns and rows, the CSC representation of a matrix $\mathbf{A}$ is equivalent to the CSR representation of the transpose $\mathbf{A}^T$. Within each row or column, the entries can be sorted by column or row index. This allows for the use of an efficient binary search when performing random access to entries specified by row and column.

The CSR and CSC sparse matrix data structures are illustrated in Fig. 2.3 for a small example matrix. Compared to COO, CSR and CSC are significantly more compact, unless the matrix is hypersparse, i.e., a low-rank matrix with many rows or columns that only contain zero values. Furthermore, SpMVs using CSR matrices are trivially parallelizable, as each row can be computed independently. For a small number of threads, similar performance can be achieved with CSC matrices by maintaining a result vector per thread and performing a final summation step. However, with rising multicore CPU core counts and on GPUs, such an approach is limited by memory cost and rising cost of the final summation step.

Another well-established sparse matrix layout that forms the basis of several newer formats is the ELLPACK-ITPACK format (ELL). ELL was introduced with ELLPACK [RB85], a system for solving elliptic boundary value problems. It was later used in the ITPACK project [KY88] to solve large sparse linear systems on vector processors. Like CSR, ELL is a row-based format. Sparse matrices $\mathbf{A} \in \mathbb{F}^{n \times m}$ are stored as a pair of 2D arrays, one for the column indices and one for the values, with n rows and $k = \max_i(\mathrm{nnz}_i(\mathbf{A}))$ columns, where $\mathrm{nnz}_i(\mathbf{A})$ is the number of non-zero entries in row i. For rows with fewer than k non-zero entries, zero-values are inserted as padding. These can be combined with any valid column index. Depending on processor architecture, using either a common column index, e.g., zero, or the index of the row itself can

result in better caching. As the number of operations and the amount of memory required depend on k, the format is ill-suited for highly irregular matrices with a large variance in the number of nonzeros per row. While it is commonly not specified, the 2D arrays are laid out in column-major order in the original Fortran implementation.

2.3. Summary

In summary, we have described GPU hardware architecture and the CUDA GPGPU programming model. In doing so, we have introduced the common GPGPU terminology used throughout the following chapters. These include the hierarchical parallel threading model using *blocks* of *threads* that execute *kernels* in groups of 32 threads called *warps*, as well as the memory hierarchy consisting of *global*, *local*, and *shared* memory in addition to multiple *cache* levels.

Furthermore, we have explained important performance considerations that result from the architecture and programming model. Most importantly, these are the issues caused when warps take *divergent* code paths and when memory accesses cannot be *coalesced*, as well as the differences in *bandwidth* and *latency* between the memory levels and the PCIe bus.

Additionally, we have described the concept of sparse matrices and the basic CSR, CSC, and ELL sparse matrix formats. These formats provide the foundation on which the novel data structures and methods introduced in Chapters 3 to 5 are built. In the following chapter, we introduce a variant of CSR to compactly represent directed incidence matrices and describe a linear algebraic framework for efficient parallel GPU processing of meshes represented using incidence matrices.

3. Mesh Processing for Volumetric Modeling

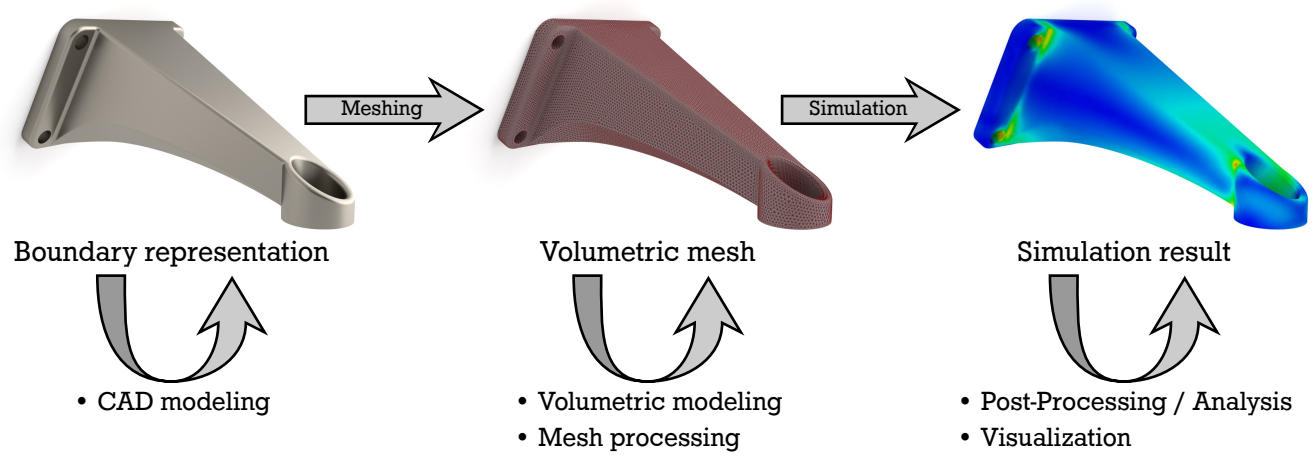


Figure 3.1.: Typical geometry pipeline in CAx tools. After modeling an object as a boundary representation (B-rep) in a CAD program, it is converted to a discrete volumetric mesh. Volumetric meshes can also be modeled directly, morphed, or processed otherwise. After simulation, physical values are associated with the mesh which can then be analyzed and visualized.

This chapter is based on the following publications:

[MAS17] **Mueller-Roemer, J. S.**, C. Altenhofen, and A. Stork. "Ternary Sparse Matrix Representation for Volumetric Mesh Subdivision and Processing on GPUs." In: *Computer Graphics Forum* 36(5) (Symposium on Geometry Processing 2017), pp. 59–69. DOI: 10.1111/cgf.13245.

[MS18] **Mueller-Roemer, J. S.** and A. Stork. "GPU-based Polynomial Finite Element Matrix Assembly for Simplex Meshes." In: *Computer Graphics Forum* 37(7) (Pacific Graphics 2018), pp. 443–454. DOI: 10.1111/cgf.13581.

The bulk of the chapter is based on the first paper [MAS17], except Section 3.4.4, which is based on the second [MS18]. Large parts of these publications are quoted verbatim with minor changes, corrections, and extensions.

3.1. Introduction

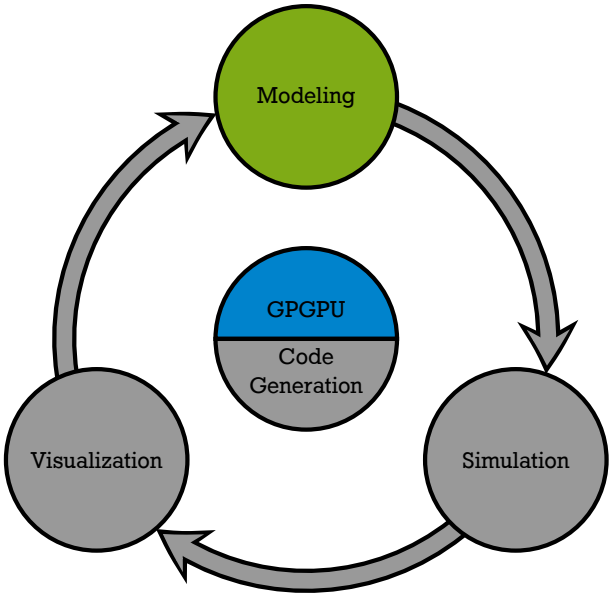


Figure 3.2.: Schematic representation of the virtual prototyping cycle, highlighting the approach used to accelerate the modeling and mesh processing step in this chapter.

In this chapter, we examine how to accelerate the modeling step of the virtual prototyping cycle shown in Fig. 3.2 on the graphics processing unit (GPU). While the use of general purpose computing on the GPU (GPGPU) for simulations themselves is widespread, mesh processing mostly remains confined to serial processing on the system processor (CPU), especially when topological changes are involved. At the same time, changes to the input mesh frequently require recomputation of several matrices and other data structures, such as spatial acceleration structures, required on the GPU. Computing these on the CPU and then transferring them to the GPU involves high synchronization and bus transfer costs and leads to significant slowdowns.

The geometry pipeline in computer-aided (CAx) tools is detailed in Fig. 3.1. While the interactive addition or modification of individual faces or cells in conventional boundary representation (B-rep), polygonal, or volumetric modeling do not lend themselves to parallelization due to the small number of entities affected, a number of other algorithms used in modeling and mesh processing can benefit significantly from parallelization. Smoothing algorithms to increase mesh quality or remove noise from scanned models belong to this category, as they potentially affect all vertices. Another category of algorithms that can benefit from parallelization are subdivision schemes. This includes both piecewise linear local refinement schemes used in h-adaptive simulation and subdivision schemes that converge to a smooth limit popular in computer graphics and multiresolution modeling. However, subdivision schemes for surfaces and volumes alike require a large amount of information about neighborhood relations. Typically, the relationships between cells, faces, edges, and vertices are required. Computing these for large meshes is often too time consuming to do on the fly, whereas storing them all drastically increases memory consumption.

With respect to the first research question posed in Chapter 1

1. **Can the GPU be used to efficiently process unstructured meshes, both polyhedral meshes in general and tetrahedral meshes in particular?** If yes, which mesh data structures and algorithms are suitable for GPU processing?

we examine how volumetric mesh modeling and processing can benefit from massively parallel GPUs hardware and data structures optimized for the GPU. We propose a data structure for meshes based on a compact, ternary sparse matrix representation of boundary operators. The proposed data structure is capable of representing both manifold and non-manifold meshes as well as meshes with mixed element types. While the concept of the presented mesh data structure is applicable both to surface meshes and volumetric meshes, we focus on volumetric meshes used in simulation.

We examine the suitability of this structure for mesh processing on the GPU. To that end, we compare several mesh processing algorithms implemented in CUDA using our data structure and compare them with their CPU counterparts implemented using Kremer et al.'s OpenVolumeMesh library [KBK13], as well as a CPU implementation of our data structure. As our approach focuses on efficient calculation and storage of neighborhood relations between mesh elements, we also implemented and analyzed the volumetric Catmull-Clark subdivision scheme as presented by Joy and MacCracken [JM96]. Combined with volumetric modeling, subdivision schemes can be used to generate simulation meshes implicitly [ASSF17] or as an alternative to isogeometric analysis based on trivariate non-uniform rational B-splines (NURBS) [BHU10b]. Additionally, we discuss how the mesh data structure can be adapted to further reduce memory requirements when simplicial meshes, i.e., triangular or tetrahedral meshes, are used. These modifications are used in Chapter 4 to accelerate the matrix assembly process in the finite element method (FEM).

The remainder of this chapter is organized as follows: in Section 3.2, we provide an overview of terminology and related work. Section 3.3 describes the basic, theoretical concepts underlying our data structure. The practical implementation of the data structure, how to adapt it for simplicial meshes, and the algorithms implemented for the evaluation are presented in Section 3.4. Section 3.5 discusses the results of our comparisons with OpenVolumeMesh (OVM). Finally, we summarize the chapter, contributions, and limitations in Section 3.6.

3.2. Related Work

In this section, we provide an overview of terminology and related work in the areas of array-based / index-based volumetric mesh data structures, mesh processing on GPUs and volumetric subdivision. For a general background on sparse matrix data structures and GPGPU, refer to Chapter 2.

3.2.1. Array-based volumetric mesh data structures

Being based on boundary operators stored as sparse matrices to represent a mesh's topology, our storage format leads to contiguously laid out arrays in memory and addressing being done via column indices and offsets. Therefore, it is more closely related to array-based (sometimes also called index-based) mesh data structures than to pointer-based data structures, such as most implementations of the half-edge / doubly connected edge list (DCEL) data structures introduced for surface meshes by Muller and Preparata [MP78] or the more recent linear cell complex data structure used in the CGAL library [Dam16].

Aside from the use of indices into arrays instead of pointers, most pointer-based data structures represent entities (edges, half-edges, faces, etc.) as individual objects, while array-based data structures spread their attributes over one or more arrays and may not store intermediate entities such as half-edges and half-faces explicitly at all. Due to the allocation of various, potentially large, objects, pointer-based data structures frequently suffer from significant memory fragmentation leading to unnecessarily high cache pressure. While advanced allocation techniques can reduce memory fragmentation to some degree, array-based data structures remain better suited for parallel and vectorized implementations due to the separate, contiguous storage of individual entity properties. Furthermore, the number of bits used for indices and offsets can be chosen or adapted freely to save memory and reduce cache pressure, while the size of pointers is dictated by the system and is currently on the order of 8 bytes (64 bits) on most systems.

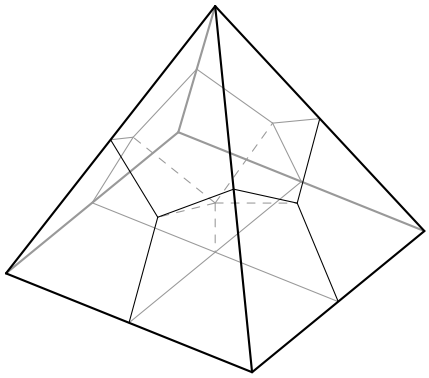


Figure 3.3.: Topology of a pyramid after Catmull-Clark subdivision without smoothing. The new cell connected to the top vertex is not hexahedral, but an octahedron with ten vertices and non-planar quadrilateral faces.

A variety of data structures for simplicial complexes and tetrahedral meshes have been proposed by authors such as De Floriani et al. and Lage et al. [DH03; DGH04; LLLV05]. However, these data structures are limited to one cell type (tetrahedra for 3D meshes) by definition, limiting their usefulness. In particular, general volumetric Catmull-Clark subdivision is not possible. While it leads to hexahedral-dominant meshes, it can also lead to non-hexahedral cells, depending on the input mesh. Although Catmull-Clark subdivision of surface meshes always leads to uniform quadrilateral meshes, volumetric subdivision cell types depend on the number of incident faces and edges to a vertex within a cell. For example, subdividing a four-sided pyramid leads to four hexahedra at the base and an octahedron with ten vertices and (non-planar) quadrilateral faces at the top, as the top vertex has four incident edges and faces, as shown in Fig. 3.3.

An overview of array-based mesh data structures by Alumbaugh and Jiao [AJ05] introduces an array-based variant of the half-edge data structure (HEDS), and its generalization to volumetric meshes, the array-based half-face data structure (array HFDS). Like other data structures derived from DCEL, it only supports manifold meshes. And while it is very compact, it introduces the concept of anchored half-faces, which require a fixed local numbering of vertex indices per cell and cell face. Therefore, the set of allowed cell types (polyhedra) must be known beforehand.

More recently, array HFDS was extended to support mixed-dimensional and non-manifold meshes (AHF) by Dyedov et al. [DRE+15]. While the restriction to manifold meshes is lifted, a priori knowledge of the set of allowed cell types is still required. Therefore, AHF's applicability in volumetric subdivision and other operations that can generate arbitrary cells is limited.

Another extension of DCEL to half-faces called OVM was proposed by Kremer et al. [KBK13]. By using arrays of handles instead of doubly linked lists and using a full hierarchy of relationships, they are able to represent general polytopal complexes, both without requiring a priori knowledge of all cell types and without requiring a mesh to be manifold. While their representation is less compact than array HFDS or AHF, we consider it to be the main competitor to our data structure due to its similarities and generality.

A linear algebraic mesh representation that bears some similarity to our approach is presented in a technical note by DiCarlo et al. [DPS14]. They store characteristic matrices that map each k-face (see Section 3.3) to its unordered vertices as a binary compressed sparse row (CSR) matrix. While boundary operators can

be derived efficiently, they can only be derived in an unoriented form, limiting the scheme's suitability to applications that do not require facet orientations.

3.2.2. Mesh processing on GPUs

Despite numerical simulation on GPUs being a widespread and well researched topic, GPGPU volumetric mesh processing is far less common. Surface mesh processing in the form of parallel mesh simplification has been examined by various authors, such as the works by DeCoro and Tatarchuk, Papageorgiou and Platis, and Odaker et al. [DT07; PP15; OKV15].

For volumetric meshes, a hybrid CPU-GPU algorithm for mesh optimization has been developed by D'Amato and Vénere [DV13]. While many operations are performed in parallel on the GPU, topological changes are performed on the CPU. They achieve speedups between 2.8× and 6.6× compared to the sequential version.

A different approach is used in Cheng et al.'s paper on parallel optimization of volumetric meshes on heterogeneous systems [CSY+15]. To improve parallelizability, they avoid topological changes entirely and only modify vertex positions. This improves GPU performance significantly, as GPUs are sensitive to irregular workloads within thread blocks and divergent control, as described in Section 2.1. The speedups achieved compared to the serial version range between 14× and 21× when using only the GPU. They also attempt processing interior vertices on the GPU and exterior vertices on the CPU, both due to their lower number and the higher control flow divergence, but only achieve speedups between 11× and 16× compared to the sequential version due to the CPU-GPU communication overhead.

During the initial review of the paper on which this chapter is based, Zayer et al. published a sparse matrix-based mesh representation for GPU mesh processing [ZSS17a]. They use a compressed sparse column (CSC) encoding of a face table, mapping each face (or cell) to its ordered vertices. Additionally, they introduce the concept of action maps to modify the sparse matrix-vector and sparse matrix-matrix operations to achieve various mesh operations. However, these require all faces or cells to be of the same type, and does not allow for general polytopal meshes.

Since the original publication of our paper, Mlakar et al. have published a preprint presenting a GPU-accelerated method for subdivision surfaces [MWS+19]. Their method is based on Zayer et al.'s data structure [ZSS17a] and therefore inherits its limitation of not supporting general polytopal meshes. While this is not a significant restriction for subdivision surfaces where an initial general subdivision step can be performed on the CPU, it significantly limits the applicability to subdivision volumes, as described in the previous section (see Section 3.2.1 and Fig. 3.3).

3.2.3. Volumetric Subdivision

Subdivision surfaces are widely used in computer graphics and computer animation, but increasingly also in computer-aided design (CAD) (see, e.g., [Ma05; SKSD14]), to create smooth models with a relatively small number of degrees of freedom. A coarse control mesh is refined iteratively by inserting new vertices, edges, and faces with every subdivision step. When this subdivision process is repeated an infinite number

of times, the geometry converges to the so-called limit surface that can also be calculated analytically for some subdivision schemes (see, e.g., [Sta98]). Certain schemes require purely triangular control meshes, such as the subdivision scheme presented by Loop [Loo87]. Others operate on quad-based meshes or are able to handle arbitrary polygons like the Catmull-Clark surface subdivision scheme [CC78].

The concept of volumetric subdivision extends the idea of subdivision surfaces by an additional dimension. Volumetric control meshes are used and in addition to vertices, edges, and faces, new cells are created when subdividing. Similar to surface subdivision schemes for triangle meshes, some volumetric subdivision schemes operate on tetrahedral control meshes, such as the methods by Chang et al. [CMQ03] or Schaefer et al. [SHW04]. These are often used for global or local refinement of simulation meshes for FEM simulation, as shown by Burkhart et al. [BHU10a].

In this chapter, we utilize the Catmull-Clark subdivision solids described by Joy and MacCracken [JM96] to showcase our approach. As an extension to Catmull-Clark subdivision surfaces, the volumetric Catmull-Clark subdivision scheme operates on control meshes with arbitrary polyhedra.

In a recent conference paper, Altenhofen et al. presented an approach to generate tetrahedral simulation meshes directly from volumetric Catmull-Clark models [ASSF17]. As Altenhofen et al. already use a GPU-based FEM solver (originally presented by Weber et al. [WMA+15]), their approach would benefit significantly from performing subdivision and mesh operations directly on the GPU.

3.3. Concept

In this section, we describe the concepts used in our data structure for efficient volumetric mesh representation on GPUs. We represent a volumetric mesh using the discrete boundary operators $\mathbf{\partial}_k$ that describe the top down relationships from each k-face to its oriented $(k-1)$-face facets. While the principles outlined in this section are applicable to d-dimensional meshes embedded in e-dimensional space, the current implementation is limited to 3D meshes in $\mathbb{R}^3$. To clarify the terminology of k-faces, a volumetric mesh in $\mathbb{R}^3$ consists of vertices (0-faces), edges (1-faces), faces (2-faces), and cells (3-faces).

The boundary operators $\mathbf{\partial}_k$ are linear operators that correspond to sparse, ternary matrices, i.e., all entries are values in $\{-1, 0, 1\}$ and most entries are zero. The concept of boundary operators (and coboundary operators mentioned below) are shared with discrete differential geometry (DDG) / discrete exterior calculus (DEC) (see, e.g., Desbrun et al.'s workshop paper on discrete differential forms [DKT06]) and originate in algebraic topology (see, e.g., Hatcher's book on algebraic topology [Hat02]). For regular polytopes, the fact that relationships between k-faces of dimensions differing by one can be represented as directed graphs or ternary incidence matrices has been documented earlier by Coxeter [Cox73].

To make use of the large body of research on sparse matrix operations on GPUs, we use a ternary compressed sparse row (TCSR) representation for $\mathbf{\partial}_2$ and $\mathbf{\partial}_3$. We only store non-zero entries, and encode the sign of the non-zero entries in the sign of the corresponding column index. Therefore, no separate value array is required. As an optimization, $\mathbf{\partial}_1$ is stored as a vector of ordered pairs instead, since edges have two vertices by definition. This removes the need for an offset array, no decoding is necessary as the sign is implicit, and the pairs can be aligned in memory to achieve coalescing using pairwise loads and stores.

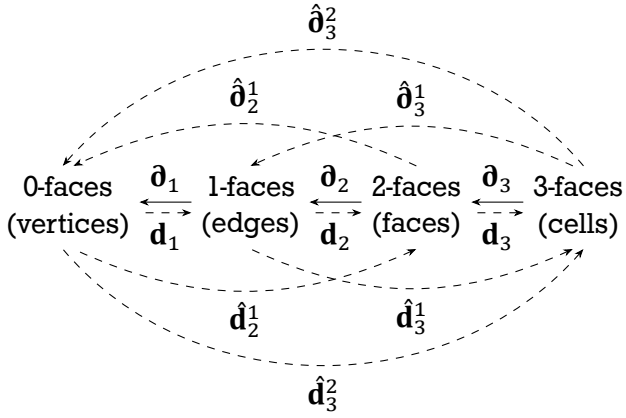


Figure 3.4.: This diagram shows how the different (co)boundary and chained operators describe the relations between k-faces. Only operators shown as solid lines are always stored. Dashed operators are computed and cached on demand.

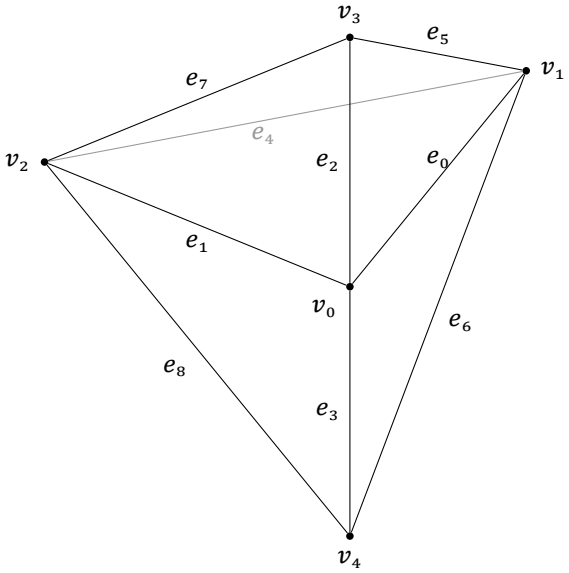
When bottom-up relationships are required, these can be computed efficiently on demand from the boundary operators $\mathbf{\partial}_k$ by computing their transpose $\mathbf{\partial}_k^T$. Therefore, efficient CSR transpose algorithms can be reused. These transposed boundary operators correspond to the boundary operators of the dual mesh or the coboundary operators $\mathbf{d}_k = \mathbf{\partial}_k^T$. While $\mathbf{\partial}_1$ is stored as a vector of ordered pairs, the dual edges $\mathbf{d}_3 = \mathbf{\partial}_3^T$ are not, as faces on the outer boundary of a mesh only have one neighboring cell and non-manifold meshes can have faces without neighboring cells. Additionally, $\mathbf{d}_1 = \mathbf{\partial}_1^T$ requires special treatment to convert from ordered pairs to TCSR. If a simpler implementation is desired, all boundary and coboundary operators can be stored in the TCSR format.

Indirect relationships such as the list of all vertices belonging to a specific face can be computed by chaining boundary operators. Direct chaining/multiplication of two consecutive boundary operators always results in a zero matrix $\mathbf{\partial}_k \mathbf{\partial}_{k-1} = \mathbf{0}$, as each well-formed k-face must be 2-manifold and each $(k-2)$-face is therefore used twice, once in its positive and once in its negative orientation. For example, each edge (1-face) within a cell (3-face) is used exactly twice, in opposing directions, by two adjacent faces (2-faces). By first taking the elementwise absolute value $\hat{\mathbf{\partial}}_{k,ij}^0 = |\mathbf{\partial}_{k,ij}|$ and then using $\max(\cdot,\cdot)$ as an additive field operator, the indirect relationships $\hat{\mathbf{\partial}}_k^n = \prod_{i=0}^n \hat{\mathbf{\partial}}_{k-i}^0$ can be computed using general sparse matrix-matrix products (SpGEMMs).

An overview of the described operators is shown in Fig. 3.4. An example mesh, along with the associated boundary operator matrices $\mathbf{\partial}_k$, is given in Fig. 3.5. While the direct, top-down boundary operators $\mathbf{\partial}_k$ are always stored, all other operators / relations $\mathbf{d}_k$, $\hat{\mathbf{\partial}}_k^n$, and $\hat{\mathbf{d}}_k^n$ are computed and cached on demand.

Due to the concepts shared with DDG and DEC, our storage format could be useful in the implementation of DEC-based simulation codes (see, e.g., [CC17]). While most treatises on DEC limit themselves to simplicial complexes (meshes consisting only of simplices), our representation based on boundary operators stored in TCSR matrices is not limited to simplicial complexes and can represent meshes with arbitrary polyhedral cells. Storage of non-manifold meshes is possible as well.

Throughout this chapter, we focus on bulk operations (processing multiple elements, instead of individual elements). This is necessary for parallelization and to fully load the GPU, which is not possible when processing individual elements or small numbers of elements. However, such operations could be useful



$$\partial_3 = \begin{array}{c} \begin{matrix} f_0 & f_1 & f_2 & f_3 & f_4 & f_5 & f_6 \end{matrix} \\ \begin{pmatrix} 1 & -1 & 0 & 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 & -1 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} c_0 \\ c_1 \end{matrix}$$

$$\partial_2 = \begin{array}{c} \begin{matrix} e_0 & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \end{matrix} \\ \begin{pmatrix} 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{matrix}$$

$$\partial_1 = \begin{array}{c} \begin{matrix} v_0 & v_1 & v_2 & v_3 & v_4 \end{matrix} \\ \begin{pmatrix} -1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 1 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} e_0 \\ e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \end{matrix}$$

Figure 3.5.: Example mesh with two tetrahedra c_0 and c_1 connected by a common face f_0, along with the resulting boundary operators. Boundary operators ∂_k describe which $(k-1)$-face facets (columns) form a k-face (rows) and their orientation (signs). Negative values indicate reversed orientation or the starting vertex in the case of edges.

even if they are slower than on CPU due to the high communication cost over the peripheral component interface express (PCIe) bus.

3.4. Implementation

In this section, we detail the implementation of our data structure and the associated algorithms. As mentioned in previous sections, we use a compact, ternary variant of CSR sparse matrices, TCSR. As outlined in Section 2.2, the basic CSR data structure consists of three 1D arrays: `columns[nnz]`, `values[nnz]`, and `offsets[nrows + 1]`, where `nnz` is the number of non-zero entries in the matrix and `nrows` is the number of rows of the matrix.

As the boundary operators only contain entries in $\{-1, 0, 1\}$, TCSR discards the `values` array and encodes the sign of the entry in `columns`:

$$\texttt{columns}_{\text{TCSR}}[i] = \begin{cases} -\texttt{columns[i]} - 1 & \text{if } \texttt{values[i]} = -1 \\ \texttt{columns[i]} & \text{if } \texttt{values[i]} = 1, \end{cases}$$

where the operation $-i-1$ corresponds to the bitwise negation of the two's-complement integer i. Zero entries are not and cannot be stored explicitly. Decoding is very simple, as every encoded column entry is negative if the value is -1. On most architectures, an optimizing compiler will replace all conditionals in encoding and decoding by combinations of `sar` (sign-extending, arithmetic shift right) and `xor` (bitwise exclusive or) instructions. This is particularly advantageous on GPUs, as control flow divergence is avoided.

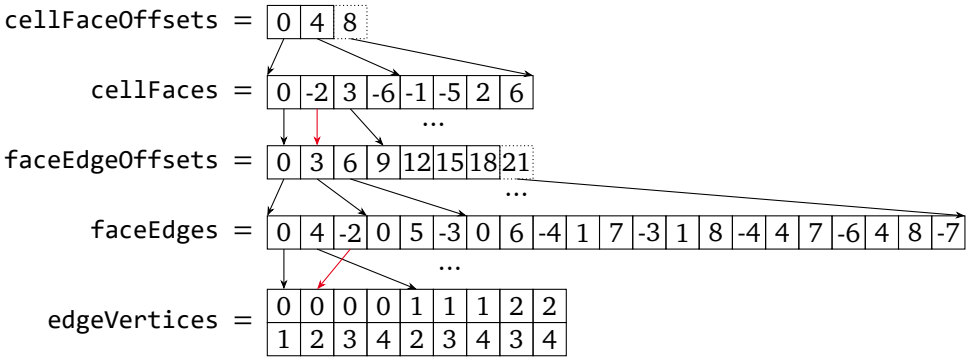


Figure 3.6.: In-memory representation of the boundary operators of the mesh in Fig. 3.5 using our TCSR encoding. Red arrows indicate that the associated facet is used in its inverted orientation. The offsets point to positions between entries, as each pair indicates a range. The borders of the last offset entries are dotted, as they do not belong to a cell or face.

On sign-magnitude architectures, a different encoding would be preferable. Our current implementation uses 32-bit integers for all offsets and encoded values.

As mentioned in Section 3.3, all operators and indirect relationships except $\mathbf{\partial}_1$ are stored in the TCSR format. The edges described by $\mathbf{\partial}_1$ are stored as ordered pairs of vertices, as the number of vertices per edge is always two. While edges could be stored in TCSR form as well, using ordered pairs removes the need for an offset array, saving memory. Furthermore, no encoding/decoding of the vertex's sign and index is necessary, and it is known a priori which of the vertices is the starting vertex. In summary, our data structure consists of the following arrays:

1. `cellFaces[ncf]`, the encoded column/value pairs of $\mathbf{\partial}_3$, where `ncf` is the number of non-zero entries of $\mathbf{\partial}_3$.
2. `cellFaceOffsets[ncells + 1]`, the row offsets of $\mathbf{\partial}_3$, where `ncells` is the number of rows of $\mathbf{\partial}_3$ and equal to the number of cells in the mesh.
3. `faceEdges[nfe]`, the encoded column/value pairs of $\mathbf{\partial}_2$, where `nfe` is the number of non-zero entries of $\mathbf{\partial}_2$.
4. `faceEdgeOffsets[nfaces + 1]`, the row offsets of $\mathbf{\partial}_2$, where `nfaces` is the number of rows of $\mathbf{\partial}_2$ and equal to the number of faces in the mesh.
5. `edgeVertices[nedges][2]`, $\mathbf{\partial}_1$ stored as ordered pairs of vertex indices. No TCSR encoding is used. No offsets are required and orientation is implicit in pair order.
6. `positions[nvertices][3]`, the positions of the vertices in $\mathbb{R}^3$, where `nvertices` is the number of vertices in the mesh.
7. The inverse and indirect relations $\mathbf{d}_k$, $\hat{\mathbf{\partial}}^n_k$, and $\hat{\mathbf{d}}^n_k$ are all stored in the TCSR format, but only allocated and computed on demand. If computed, they are cached in our data structure for reuse until invalidated by mutable access to any operator $\mathbf{\partial}_k$ on which they depend, or explicitly purged by the user.

The in-memory representation of the example mesh's boundary operators in Fig. 3.5 is illustrated in Fig. 3.6. While the basic data structure does not contain additional properties beyond vertex positions,

Listing 3.1: Pseudocode for the ternary CSR to ternary CSC transposition procedure. The individual subprocedures are explained in the text. $\bigoplus$ denotes array concatenation.

```
def TCSRtoTCSC(M):
    counts := 0                                          # M.ncols zeros
    for o in [0..M.nnz):                                 # in parallel
        c, v := DecodeTCV(M.columns[o])
        entryOffsets[o] := AtomicInc(counts[c])
    R.offsets := [0] ⊕ InclusiveScan(counts)
    for r in [0..M.nrows):                               # in parallel
        for o in [M.offsets[r]..M.offsets[r + 1]):
            c, v := DecodeTCV(M.columns[o])
            e := EncodeTCV(r, v)
            o := R.offsets[c] + entryOffsets[o]
            R.columns[0] := e
    R := SortRows(R)                                     # optional, for determinism
    return R
```

such properties can be added to any operator by using supplementary value arrays for the various TCSR matrices or for the desired entities.

3.4.1. Coboundary Operators and Basic Queries

To compute the coboundary operator matrices from the boundary operators, the boundary matrices must be transposed. Transposition of CSR matrices is equivalent to conversion to the CSC format, and reinterpreting columns as rows and vice versa. The algorithm `TCSRtoTCSC` used to transpose the matrices follows the count-scan-fill pattern that underlies many of the procedures outlined in this thesis and is given in Listing 3.1.

The `AtomicInc` subprocedure used is an atomic increment operation that returns the old value before addition. As many separate counters are used, contention is minimal. `InclusiveScan` is a parallel, inclusive cumulative sum, i.e., the sequence $[3, 1, 0, 2]$ is transformed into $[3, 4, 4, 6]$ in parallel, as implemented in libraries such as Thrust [BH15]. The `EncodeTCV` operation is the encoding described in the previous section, and `DecodeTCV` is its inverse. At the end of `TCSRtoTCSC`, all rows are sorted (in parallel) by `SortRows` according to their decoded column index to achieve a deterministic order. In our current implementation, each thread of the `SortRows` kernel performs a sequential sort of a single row.

In summary, the number of outputs is first *counted* (while storing local offsets, if necessary), then an inclusive *scan* is performed to compute the offsets in the result from the counts. Finally, the output is *filled* with the desired values, for which the offsets are now known, allowing the operation to be performed in parallel. The transposition of edges uses a modified version of `TCSRtoTCSC`, where the even indices correspond to positive values and odd indices to negative values. The offsets of the input are known as well, as each row has exactly two offsets.

As mentioned in Section 3.3, the computation of indirect indices corresponds to an SpGEMM operation with modified operations. While the GraphBLAS forum, as founded by Mattson et al. [MBB+13], is building a library of graph operations in the language of linear algebra, additional assumptions can be made for well formed meshes. Therefore, we use specialized algorithms instead of a modified SpGEMM proce-

dure. A more concise, albeit likely less performant, implementation of mesh operations may be possible with GPU-based implementations of GraphBLAS, such as GBTL-CUDA or GraphBLAST [ZZL+16; Yan19]. However, compact encodings of ternary matrices are currently not supported by GraphBLAS, as the representation of incidence matrices as pairs of boolean matrices, one for incoming and one for outgoing edges, is preferred instead [KAB+16].

$\hat{\mathbf{\partial}}_2^1$ (`faceVertices`) can be computed without previously counting the vertices of each face, as each face forms a closed loop. For $\hat{\mathbf{\partial}}_3^1$ (`cellEdges`), the number of unique edges per cell can be computed as the sum of edges of each cell divided by two, as each edge is used once in each direction. Following the count-scan-fill pattern and after performing an inclusive scan on these counts to compute the offsets, the output columns can be allocated according to the last offset entry. Finally, a last pass over each cell's face is performed and positively oriented edge uses (`cellFaceOrientation`$\cdot$`faceEdgeOrientation` $= 1$) are stored in the column array.

Due to the double indirection in $\hat{\mathbf{\partial}}_3^2$, orientation can no longer be used to discard multiple references. Therefore, the number of edges per cell is computed as above without division as each edge has two vertices. After the scan and fill operations, the per-cell vertex arrays are sorted in memory and duplicate entries are moved to the back and set to a negative number while the new sizes are written to a second count array. Finally, the counts are accumulated again and the vertex array is compacted to remove negative entries using Thrust. As the aforementioned assumptions only hold for the primal mesh, the indirect bottom-up relationships are computed from their top-down duals $\hat{\mathbf{d}}_k^n = \hat{\mathbf{\partial}}_k^{n,T}$.

3.4.2. Boundary Extraction and Laplacian Smoothing

Using the $\mathbf{d}_3$ coboundary operator that maps faces to cells, the set of boundary faces is trivial to compute. In a manifold mesh, each boundary face has exactly one cell that uses it, therefore only the distance of the offsets within $\mathbf{d}_3$ is required to create a boolean mask of boundary faces. To find the orientation of the boundary face, the orientation of the cell is read from $\mathbf{d}_3$ and inverted.

For non-manifold meshes, a different approach is required, as a face may have no cells that use it causing both half-faces to be part of the boundary set. Additionally, a single orientation may be used more than once causing a boundary face to have two or more cells using it. However, even in the non-manifold case, all necessary information is present in $\mathbf{d}_3$. To mark boundary edges and vertices, an additional pass over $\mathbf{d}_2$ and $\hat{\mathbf{d}}_2^1$ is required, respectively. With the boolean markers given, extracting boundary elements becomes a simple compaction.

Laplacian smoothing of inner vertices also requires the set of boundary vertices, as these should remain unaffected. This mask is computed once, then smoothing is performed iteratively. Each smoothing iteration sets a vertex's position to the average of the positions of its neighbors. To perform this smoothing in parallel, separate input and output arrays of positions are used that are swapped between iterations using an efficient pointer exchange. The set of neighbors is determined by iterating over the corresponding entries in $\mathbf{\partial}_1$ and $\mathbf{d}_1$, which is also computed once beforehand. Which entry of $\mathbf{\partial}_1$ has to be read is determined by the sign of the entry in $\mathbf{d}_1$.

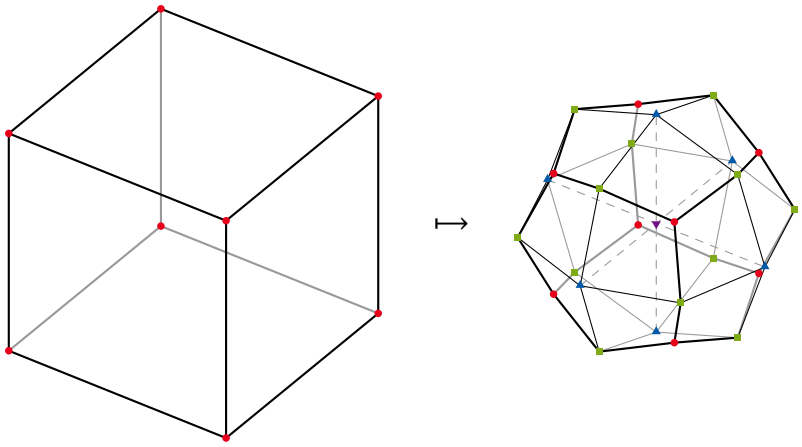


Figure 3.7.: Example of the application of the volumetric Catmull-Clark subdivision rules to a mesh consisting of a single cell. Original vertices are shown as red circles, edge points as green squares, face points as blue triangles, and the cell point as a purple upside-down triangle. Thick lines correspond to (split) original edges, while thin lines correspond to added face- (solid) and cell-edges (dashed).

3.4.3. Catmull-Clark Subdivision

While boundary extraction and Laplacian smoothing only use but do not affect or create topological information, volumetric Catmull-Clark subdivision requires the creation of new vertices, edges, faces, and cells. We briefly summarize the volumetric Catmull-Clark subdivision rules, as defined by Joy and MacCracken [JM96] and modified by Burkhart et al. [BHU10b].:

1. For each cell, add a cell point at its centroid $\mathbf{p}_C$.
2. For each face, add a face point at $\mathbf{p}_F = \frac{\mathbf{p}_{C,\text{avg}} + \hat{\mathbf{p}}_F}{2}$, where $\mathbf{p}_{C,\text{avg}}$ is the average of the cell points of the two incident cells and $\hat{\mathbf{p}}_F$ is the face's centroid.
3. Split each edge, adding an edge point at $\mathbf{p}_E = \frac{\mathbf{p}_{C,\text{avg}} + 2\mathbf{p}_{F,\text{avg}} + (n-3)\hat{\mathbf{p}}_E}{n}$, where $\mathbf{p}_{C,\text{avg}}$ and $\mathbf{p}_{F,\text{avg}}$ are the averages of cell and face points of incident cells and faces, respectively, $\hat{\mathbf{p}}_E$ is the edge midpoint, and n is the number of incident faces.
4. For each cell, connect its cell point to all its face points and connect all its face points to all incident edge points, creating corresponding faces and splitting the cell.
5. Move each original vertex $\hat{\mathbf{p}}_V$ to its new location $\mathbf{p}_V = \frac{\mathbf{p}_{C,\text{avg}} + 3\mathbf{p}_{F,\text{avg}} + 3\mathbf{p}_{E,\text{avg}} + \hat{\mathbf{p}}_V}{8}$, where $\mathbf{p}_{C,\text{avg}}$, $\mathbf{p}_{F,\text{avg}}$, and $\mathbf{p}_{E,\text{avg}}$ are the averages of the cell, face and edge points of all adjacent cells, faces, and edges.

For boundary vertices, edges, and faces, the positions are determined according to the Catmull-Clark subdivision rules for surfaces [CC78]:

1. Boundary face points are placed at the faces' centroids.
2. Boundary edge points are placed at $\mathbf{p}_E = \frac{\mathbf{p}_{F,\text{avg}} + \hat{\mathbf{p}}_E}{2}$, where $\mathbf{p}_{F,\text{avg}}$ is the average of the two incident boundary face points and $\hat{\mathbf{p}}_E$ is the edge midpoint.
3. Boundary vertices are moved to $\mathbf{p}_V = \frac{\mathbf{p}_{F,\text{avg}} + 2\mathbf{p}_{E,\text{avg}} + (n-3)\hat{\mathbf{p}}_V}{n}$, where $\mathbf{p}_{F,\text{avg}}$ and $\mathbf{p}_{E,\text{avg}}$ are the averages of the face and edge points of all adjacent boundary faces and edges, respectively, $\hat{\mathbf{p}}_V$ is the original vertex position, and n is the number of incident boundary faces.

An example of the application of these rules for a mesh consisting of a single cell is shown in Fig. 3.7.

In the first step, the number of new entities is determined to allocate the output mesh's arrays:

$$|V_{CC}| = |V| + |E| + |F| + |C| \tag{3.1}$$

is the new number of vertices and equals the sum of the old vertex, edge, face, and cell counts. Equivalently, these are the number of nonzeros in the identity matrices mapping each entity to itself. Each existing vertex is maintained, while each edge, face, and cell are mapped to a new point at the center of the corresponding entity before smoothing.

$$|E_{CC}| = \text{nnz}(\mathbf{\partial}_1) + \text{nnz}(\mathbf{\partial}_2) + \text{nnz}(\mathbf{\partial}_3) \tag{3.2}$$

is the new number of edges and equals the total number of nonzeros (nnz) in all boundary operators. Referring back to Fig. 3.3, it can be observed that each edge is split, i.e., there is one "edge-edge" for each vertex of each edge. Additionally, for each face in the mesh, a "face-edge" is added for each edge of each face, connecting the new edge vertices to the new face vertex. Finally, a "cell-edge" is added for each face of each cell, connecting the new face vertices to the new cell vertex.

$$|F_{CC}| = \text{nnz}(\hat{\mathbf{\partial}}_2^1) + \text{nnz}(\hat{\mathbf{\partial}}_3^1) \tag{3.3}$$

is the new number of faces and adds another level of indirection. Each face is split along the new "face-edges" connecting these with the "edge-edges" of edges that share the face's vertex. Similarly, "cell-faces" are created for the "cell-edges" and "face-edges" that share the cell's edge.

$$|C_{CC}| = \text{nnz}(\hat{\mathbf{\partial}}_3^2) \tag{3.4}$$

is the new number of cells. One for each vertex within each cell. Each "cell-cell" is enclosed by the "face-faces" corresponding to that vertex and the "cell-faces" connecting these back to the cell point. In summary, all element counts can be determined directly from the TCSR matrices, and the offsets of these matrices can directly be used as element offsets in the new lists of elements. However, all top-down relationships must be computed before subdivision.

The positions of the new cell points are computed as

$$\mathbf{p}_{CC,C} = \hat{\mathbf{\partial}}_3^2\mathbf{p} \cdot \text{diag}(\hat{\mathbf{\partial}}_3^2\mathbf{1})^{-1} = \text{Avg}_{\hat{\mathbf{\partial}}_3^2}(\mathbf{p}), \tag{3.5}$$

where $\mathbf{p}$ is the vector of 3-vectors of input vertex positions, $\mathbf{1}$ is a vector of scalar 1-values, and $\text{diag}(\mathbf{x})$ creates a square matrix from a vector $\mathbf{x}$ of diagonal entries. This corresponds to the cell centers, defined as a simple average of the positions of all vertices used by each cell. For the other positions, additional helper positions

$$\hat{\mathbf{p}}_F = \text{Avg}_{\hat{\mathbf{\partial}}_2^1}(\mathbf{p}) \tag{3.6}$$

and

$$\hat{\mathbf{p}}_E = \text{Avg}_{\hat{\mathbf{\partial}}_1^0}(\mathbf{p}) \tag{3.7}$$

are computed. The matrix multiplication and division by the number of entries in Eqs. (3.5) to (3.7) are fused in a single GPU kernel.

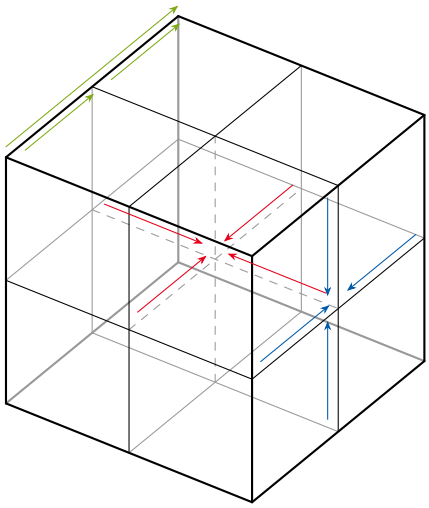


Figure 3.8.: Catmull-Clark edge orientations. "Edge-edges" (green) use the orientation of the original edge. "Face-edges" (blue) and "cell-edges" (red) point towards the corresponding face and edge points, respectively.

The final face point positions also require the calculation of the boundary as described in Section 3.4.2. While the topological operations require all top-down relationships, the computation of the final positions requires all bottom-up relationships.

$$\mathbf{p}_{CC,F} = \begin{cases} \hat{\mathbf{p}}_F & \text{if boundary} \\ \dfrac{\hat{\mathbf{p}}_F + \mathrm{Avg}_{\hat{\mathbf{d}}_3^0}(\mathbf{p}_{CC,C})}{2} & \text{otherwise,} \end{cases} \tag{3.8}$$

which corresponds to the centers of the original faces for faces on the boundary (as in Catmull-Clark subdivision surfaces) and to a weighted average of the face center and the centers of the two neighboring cells for non-boundary faces. The individual faces are processed using one thread per face. The edge point and new vertex positions are computed in a similar manner as weighted averages of their centers and the center points of all connected higher-level facets (i.e., edges, faces and cells).

After computing all positions, the new mesh's topology must be constructed. This is again done for each set of entities separately. First, edges are created with the orientations shown in Fig. 3.8. Edges are split according to the following rule:

$$e : (v_1, v_2) \rightarrow \begin{matrix} 2e : (v_1, e + |V|) \\ 2e + 1 : (e + |V|, v_2) \end{matrix}, \tag{3.9}$$

where e is the index of the edge connecting the vertices with the indices v_1 and v_2. Face edges are created as follows:

$$f : (o_1 : e_1, \ldots, o_n : e_n) \rightarrow o_i + \mathrm{nnz}(\mathbf{\partial}_1) : (e_i + |V|, f + |V| + |E|)|_{i=1}^{n}, \tag{3.10}$$

where f is the index of the face consisting of the edges e_1, …, e_n with the entries at the offsets o_1, …, o_n. Cell edges are created analogously.

As for the edges, the size of each face is known a priori. After Catmull-Clark subdivision every face is a quadrilateral. For the "face-faces", the orientation of the original face is kept, as shown in Fig. 3.9. The orientation of the "cell-faces", however, cannot be derived from the original mesh directly. In order to

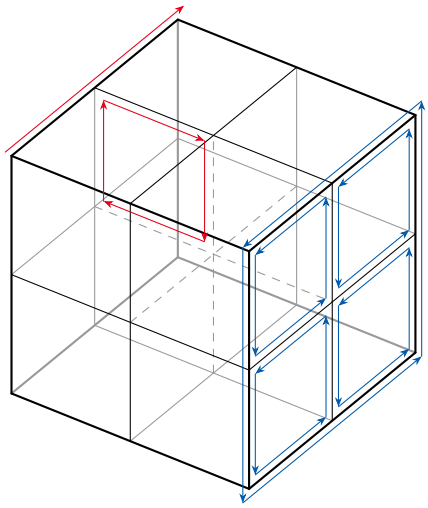


Figure 3.9.: Catmull-Clark face orientations. "Face-faces" (blue) are oriented in the same direction as the global orientation of the originating face. "Cell-faces" (red) are oriented according to the global orientation of the originating edge.

orient them consistently, we decided to define the "cell-face" in a way that its normal vector is oriented in the global direction of the original edge whose edge point is used in the current "cell-face". This is done using only the topological information, in particular the orientation in which the two faces that reference it use the edge, and not the positions of the mesh.

While the number of resulting cells can be determined directly, including their index, each "cell-cell" has a varying number of indices (Figure 3.3 shows an example with a non-hexahedral cell). Therefore, the count-scan-fill pattern is used again. To count the number of "cell-faces" per "cell-cell", we iterate over the entries of $\hat{\mathbf{\partial}}_3^2$ for each cell in parallel and count the number of faces within each cell that use the vertex. This number is additionally multiplied by two to account for the "edge-faces" belonging to the edges between the faces.

After the scan, the output mesh's `cellFaces` array can be filled. As during counting, we iterate over $\hat{\mathbf{\partial}}_3^2$ and find the cell's faces and edges that use the vertex. The "face-faces" are used in the same orientation as the corresponding faces. The orientation of "cell-faces" are determined by comparing the starting vertex with the current vertex index.

3.4.4. Simplicial Meshes

As simplex meshes are common in simulation and geometry processing, we additionally present a specialized variant of the TCSR data structure for simplex meshes that is used in Chapter 4. While representing the topology of arbitrary polyhedral meshes requires storing all coboundary operators $\mathbf{\partial}_k$, the cell-to-vertex operator $\hat{\mathbf{\partial}}_d^{d-1}$ is sufficient for d-dimensional homogeneous simplicial complexes when a prescribed column index order is used. Homogeneous or pure simplicial complexes are simplicial meshes in which every k-simplex or -face with $k < d$ is part of a $(k+1)$-simplex and therefore of a top-level d-simplex (cf. [Hat02]). The binary matrix $\hat{\mathbf{\partial}}_d^{d-1}$ has a constant number of nonzeros per row. Therefore, the orientation of the top-level cells can be encoded in the order of indices in an $n \times (d+1)$ indexed face/cell array, where n is the number of top-level cells, analogous to the face table use by Zayer et al. [ZSS17a].

$$\hat{\mathbf{\partial}}_3^2 = \begin{matrix} v_0 & v_1 & v_2 & v_3 & v_4 \end{matrix} \begin{pmatrix} 1_0 & 1_1 & 1_2 & 1_3 & 0 \\ 1_0 & 1_2 & 1_1 & 0 & 1_3 \end{pmatrix} \begin{matrix} c_0 \\ c_1 \end{matrix} \quad \texttt{cellVertices} = \begin{array}{|c|c|c|c|} \hline 0 & 1 & 2 & 3 \\ \hline 0 & 2 & 1 & 4 \\ \hline \end{array}$$

Figure 3.10.: Cell-to-vertex operator $\hat{\mathbf{\partial}}_3^2$ and compressed simplex mesh encoding of the example mesh shown in Fig. 3.5. The indices on the non-zero entries of $\hat{\mathbf{\partial}}_3^2$ indicate the order of nonzeros. Compared to standard TCSR encoding in Fig. 3.6, the minimum memory requirements to represent the mesh's topology are significantly reduced.

Due to the focus on volumetric, tetrahedral meshes, we call the array encoding $\hat{\mathbf{\partial}}_3^2$ `cellVertices`. We store the vertex indices in row-major order, where each cell represents a row, as the four indices fit into a single 16-byte memory transfer. For triangle meshes, storage in column-major order may be preferable to achieve coalescing. Alternatively, warp-transposed loads that transpose the data in shared memory can be used. This primitive is provided by libraries such as CUB [Mer18]. An example encoding of the mesh given in Fig. 3.5 is shown in Fig. 3.10.

In simplex meshes, all top-down operators $\mathbf{\partial}_k$ and $\hat{\mathbf{\partial}}_k^n$ have a constant number of nonzeros per row, as any k-simplex contains

$$\binom{k+1}{l+1} \tag{3.11}$$

l-simplices because any k-simplex has $k+1$ vertices and any combination of $l+1$ of its vertices forms an l-simplex. Therefore, the numbers of nonzeros per row can be computed as

$$\mathrm{nnz}_i(\mathbf{\partial}_k) = \binom{k+1}{k} = k+1 \tag{3.12}$$

and

$$\mathrm{nnz}_i(\hat{\mathbf{\partial}}_k^n) = \binom{k+1}{k-n}, \tag{3.13}$$

respectively, and the offset arrays can be omitted. Furthermore, this indicates a simple method for determining all other top-down operators from $\hat{\mathbf{\partial}}_d^{d-1}$. The other simplex-to-vertex operators $\hat{\mathbf{\partial}}_k^{k-1}$ (and $\mathbf{\partial}_1$) for $k < d$ are easily derived from `cellVertices` by filling a larger array with the sorted tuples of all combinations of $k+1$ vertices of each cell, then sorting that array of tuples and removing duplicate entries. Sorting and removal of duplicates are standard parallel primitives and efficiently implemented in libraries such as Thrust or CUB [BH15; Mer18].

As the resulting arrays are sorted, the other top-down operators can be efficiently determined by performing an $\mathcal{O}(\log_2 n)$ binary search for the sorted tuple of vertex indices of the corresponding facet or subelement of each element in parallel. For the boundary operators $\mathbf{\partial}_k$, where $k \neq 1$, the orientation of the facets must be determined as well. The orientation o of each facet can be computed as

$$o = (-1)^{i+s}, \tag{3.14}$$

where i is the lexicographical index of the combination and s is the number of swaps required to sort the global vertex indices of the combination. For the second tetrahedron c_1 of the example mesh (see Figs. 3.5 and 3.10) this results in:

Combination	i	Global	Sorted	s	o
(0, 1, 2)	0	(0, 2, 1)	(0, 1, 2)	1	-1
(0, 1, 3)	1	(0, 2, 4)	(0, 2, 4)	0	-1
(0, 2, 3)	2	(0, 1, 4)	(0, 1, 4)	0	+1
(1, 2, 3)	3	(2, 1, 4)	(1, 2, 4)	1	+1

For the bottom-up relations, the number of nonzeros per row is irregular and no meaningful order beyond sorting by column index is defined. Therefore, $\mathbf{d}_k$ and $\hat{\mathbf{d}}_k^n$ are stored in the same manner as for general meshes and computed via transposition as described in Section 3.4.1. The only significant difference is that the offsets of the input matrix are determined implicitly.

3.5. Results

All benchmarks were compiled using Microsoft Visual Studio 2015 Update 3 and NVIDIA CUDA 8.0. The measurements were performed on a desktop PC with Windows 10 64-bit, an Intel Core i7-6700 CPU, 32 GiB of DDR4-2133 main memory and a GeForce GTX 1080 GPU with 8 GiB of GDDR5X memory. All benchmarks have been repeated 20 times and averaged. No outliers were observed. Dynamic updating of bottom-up connectivity was disabled in all OVM benchmarks, as recommended by Kremer et al. [KBK13]. In addition to the GPU version of our data structure and algorithms described in Section 3.4, a CPU version that replaces all parallel algorithms by their direct, serial equivalents has been implemented and evaluated.

Table 3.1.: Mesh sizes in elements and bytes for the armadillo, dragon and bunny models on different subdivision levels. The initial meshes are tetrahedral meshes, after subdivision they are hexahedral meshes. Calculated mesh sizes in bytes are given using OVM or our data structure. All suffixes on byte sizes are binary suffixes (2^{10}, 2^{20}, ...).

Mesh	Lvl.	Vertices	Edges	Faces	Cells	OVM	Ours
Armadillo	0	1.01 k	5.08 k	7.23 k	3.16 k	348 Ki	226 Ki
	1	16.4 k	44.4 k	40.6 k	12.6 k	2.25 Mi	1.64 Mi
	2	114 k	327 k	314 k	101 k	17.3 Mi	12.5 Mi
	3	857 k	2.52 M	2.47 M	809 k	135 Mi	97.7 Mi
	4	6.65 M	19.8 M	19.6 M	6.47 M	1.05 Gi	773 Mi
Bunny	0	10.7 k	64.7 k	104 k	50 k	4.92 Mi	3.16 Mi
	1	229 k	641 k	612 k	200 k	33.8 Mi	24.5 Mi
	2	1.68 M	4.93 M	4.85 M	1.60 M	266 Mi	192 Mi
	3	13.1 M	38.9 M	38.6 M	12.8 M	2.06 Gi	1.49 Gi
Dragon	0	232 k	1.21 M	1.80 M	825 k	85.1 Mi	55.0 Mi
	1	4.06 M	11.1 M	10.3 M	3.30 M	573 Mi	417 Mi

Table 3.1 (level 0) lists the sizes of the three tetrahedral meshes used in our evaluation, as well as the memory requirements of all meshes using either OVM or our data structure. While similar information is stored, our TCSR-based representation is more compact than OVM and uses between 27% and 36% less memory. Figure 3.11 shows a sliced view of each model to give the reader a better impression of the resolutions of the individual meshes.

In the following, the results are organized in subsections in the same order as in Section 3.4. The results given in Sections 3.5.1 and 3.5.2 all refer to the input meshes (level 0 in Table 3.1). While the input

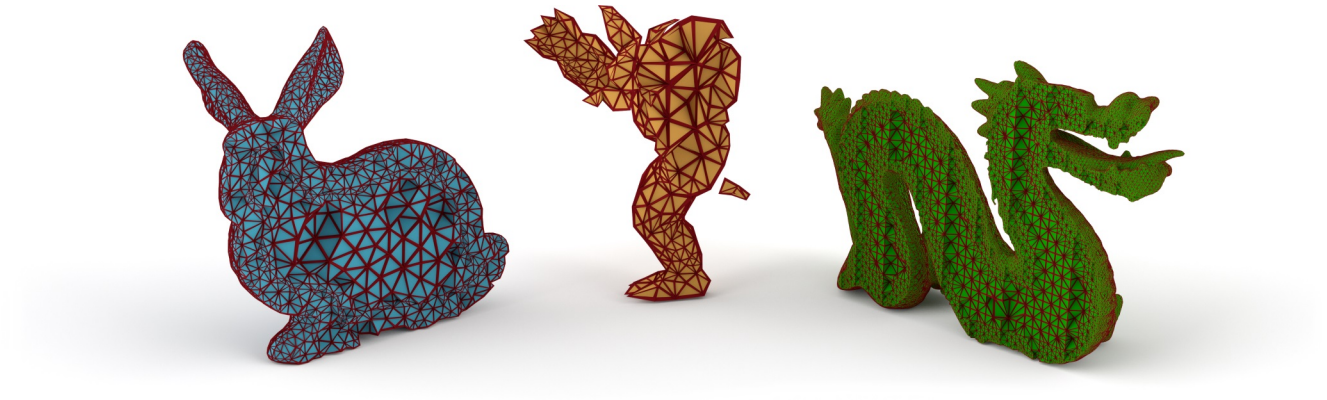


Figure 3.11.: Sliced views of the three models used in the comparison give an impression of the initial mesh resolutions. These models were chosen due to the initial publication in the field of computer graphics and due to the range of resolutions covered. The armadillo has the lowest resolution, followed by the bunny, while the dragon has the highest resolution of the three.

meshes are tetrahedral, the specialized simplicial mesh data structure presented in Section 3.4.4 is not used in this evaluation.

3.5.1. Coboundary Operators and Basic Queries

To compare the computation times of the coboundary operators $\mathbf{d}_k$, we first copied the OVM mesh to a new mesh instance with disabled bottom-up indices. We then measured the time taken to enable each individual bottom-up index. For our mesh data structure, we equivalently measured the times taken to transpose the boundary operators $\boldsymbol{\partial}_k$. For the indirect relationships $\hat{\boldsymbol{\partial}}_k^n$, we compared computing them using our data structure to iterating over the mesh using the corresponding iterator in OVM while only incrementing a counter.

Table 3.2.: Benchmark results for the calculation of inverse relations as well as indirect relations. For OVM, we used the iterators provided by the library. All times are given in milliseconds.

Operation	Method	Mesh		
		Armadillo	Bunny	Dragon
Transpose Cells	CPU	0.20	2.49	114
	GPU	0.22	0.70	14.0
	OVM	19.6	372	6788
Transpose Faces	CPU	0.26	3.07	65.6
	GPU	0.20	1.17	7.02
	OVM	3.19	56.3	1424
Transpose Edges	CPU	0.16	1.71	35.9
	GPU	0.72	0.58	4.11
	OVM	0.62	5.73	184
Cell Edges	CPU	3.08	46.7	935
	GPU	0.98	22.2	237
	OVM	1.91	34.1	691
Cell Vertices	CPU	3.74	57.2	1236
	GPU	2.72	25.1	261
	OVM	3.90	65.5	1369
Face Vertices	CPU	0.05	0.85	17.3
	GPU	0.12	0.44	1.88
	OVM	2.08	26.3	558

The results of this comparison are shown in Table 3.2. While the speeds for the smallest armadillo model are comparable between OVM and our GPU implementation, for all but face transposition, face vertex

iteration, and cell transposition, where we measured speedups of 16×, 18×, and 89×, respectively. This may be due to a bug or inefficient implementation in OVM. Only the transpose of the edges of the smallest model shows a slight slowdown of 0.86×. For the large dragon model, significant speedups between 2.9× and 531× are achieved in all cases. Comparing OVM with our CPU implementation, good speedups up to 149× are achieved in all cases, even for small models with speedups of up to 98×. The only exception is the computation of cell edges, with a slight slowdown between 0.62× and 0.74×. As expected, the GPU version is significantly faster than the other variants for sufficiently large models.

3.5.2. Boundary Extraction and Laplacian Smoothing

In the case of boundary extraction, we compared building the list of boundary face indices to using the provided boundary face iterator in OVM while only incrementing a counter. Laplacian smoothing in OVM was implemented by repeatedly iterating over all vertices, checking if they are on the boundary, and averaging their neighbor positions. As in the case of boundary extraction, we use the provided iterators. In this case, the vertex iterator to enumerate all vertices and the vertex over half edge iterator to enumerate each vertex's neighbors.

Table 3.3.: Benchmark results for the extraction of the outer surface as well as Laplacian smoothing of inner vertices. All times are given in milliseconds.

Operation	Method	Mesh		
		Armadillo	Bunny	Dragon
Surface Extraction	CPU	0.03	0.28	5.13
	GPU	1.13	1.15	2.83
	OVM	0.15	1.21	22.7
Laplacian Smoothing	CPU	0.54	21.4	860
	GPU	3.58	4.48	116
	OVM	19.0	1026	33 474

The results of these comparisons are given in Table 3.3. Laplacian smoothing is much faster in our GPU implementation and shows a speedup of 5.3× to 289× for the smallest and largest models, respectively. Our CPU implementation achieves a speedup between 35× and 48×, and is not significantly affected by mesh size. The GPU speedup for boundary computation is 8× for the largest model, but exhibits a slowdown of 0.14× on the smallest model. However, this is not surprising as the boundary consists of a very small number of triangles on the smallest model and is insufficient to load the GPU fully. The speedup of the CPU implementation is again mostly independent of mesh size and between 4.3× and 5×. As in the case of the coboundary operators and basic queries, this demonstrates that our new, compact encoding can provide benefits on both GPU and CPU.

3.5.3. Catmull-Clark Subdivision

For volumetric Catmull-Clark subdivision, a direct comparison between subdivision using our data structure and subdivision using OVM and its iterators has been performed. As OVM was significantly slower, only the first subdivision level was computed. For our data structure, we also measure the time taken to compute the full set of indirect and bottom-up relationships separately. The sizes of the subdivided meshes are given in Table 3.1. A visualization of the subdivision of the armadillo model is shown in Fig. 3.12.

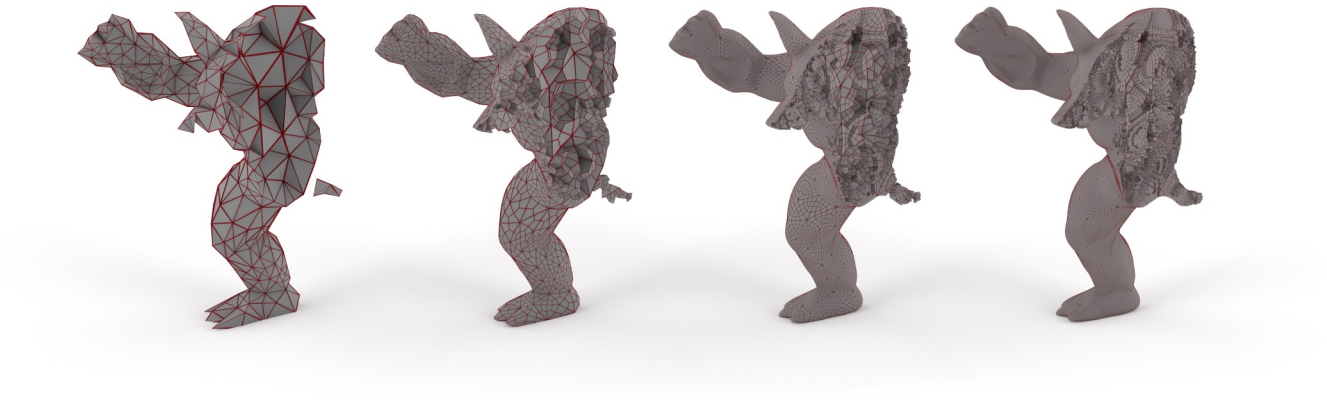


Figure 3.12.: Sliced view of the armadillo model at subdivision levels 0 to 3, from left to right, showing the inner structure of the mesh. As in Fig. 3.11, the armadillo was chosen due to the initial publication in the field of computer graphics.

Table 3.4.: Benchmark results for volumetric subdivision of the three meshes. For OVM, only one level of subdivision has been computed. For our approach, we also measured the time required for pre-calculating the indirect relations needed for the subdivision process. All times are given in milliseconds.

Lvl.	Operation	Mesh		
		Armadillo	Bunny	Dragon
1	CPU Pre.	7.50	120	2692
	CPU Total	13.4	159	3844
	GPU Pre.	6.33	57.3	553
	GPU Total	8.71	66.6	754
	OVM	392	7058	125 625
2	CPU Pre.	33.9	611	—
	CPU Total	60.7	1010	—
	GPU Pre.	13.8	173	—
	GPU Total	18.3	236	—
3	CPU Pre.	294	4847	—
	CPU Total	487	8013	—
	GPU Pre.	93.1	1036	—
	GPU Total	121	1401	—
4	CPU Pre.	2333	—	—
	CPU Total	3868	—	—
	GPU Pre.	537	—	—
	GPU Total	717	—	—

The results of this comparison are given in Table 3.4. Using our GPU version, the speedups for the first subdivision level range from 45× to 166×, and even the large dragon model takes less than a second for subdivision. The CPU version achieves speedups between 47× and 59×. The additional subdivision levels (limited by the two-second GPU kernel timeout on Windows), show promising results as well.

3.6. Summary

In summary, we have described a novel data structure for mesh representation based on a compact ternary sparse matrix encoding to store boundary operators. The novel ternary compressed sparse row (TCSR) format enables very compact storage of sparse ternary matrices such as incidence matrices and boundary operators, while remaining efficient to encode, decode, and process in parallel. The boundary-operator-based representation can describe general, potentially non-manifold, polyhedral meshes. While we developed and evaluated the data structure for volumetric meshes, all concepts can be applied directly to polygonal meshes as well. Additionally, we presented a specialization for homogeneous simplicial complexes, allowing for even more compact storage when triangular or tetrahedral meshes are sufficient.

The data structure has been implemented for volumetric meshes and we have examined the suitability of this data structure for volumetric mesh processing on GPUs by implementing computation of indirect and reverse relationships, Laplacian smoothing of inner vertices, and volumetric Catmull-Clark subdivision. These implementations have been compared to their equivalents using OVM, achieving speedups between $3\times$ and $531\times$, more than two orders of magnitude, for sufficiently large meshes. At the same time, memory requirements are reduced by up to 36%.

With respect to the research questions posed in Chapter 1, this chapter provides an answer to the first sub-question:

1. **Can the GPU be used to efficiently process unstructured meshes, both polyhedral meshes in general and tetrahedral meshes in particular?** If yes, which mesh data structures and algorithms are suitable for GPU processing?

We have shown that both arbitrary polyhedral meshes and simplex meshes can be efficiently represented and processed on the GPU, by compactly encoding orientation information in the sign bits or the order of the column indices in a CSR-like data structure and using efficient parallel map and scan primitives. With speedups of more than two orders of magnitude on operations such as smoothing and explicit subdivision used in multiresolution editing and mesh processing for simulation, the question can be answered with a clear "yes." In the following chapter, we will examine how these data structures, particularly the simplex mesh data structure presented in Section 3.4.4, can be used to accelerate the assembly step of the simulation process.

Due to the use of offset arrays and indices without any additional level of indirection, the removal of individual elements is difficult. As pointed out in the introduction, however, the addition or modification of individual elements generally does not lend itself to efficient parallelization. When performing addition or removal of a large number of elements, e.g., when performing subdivision, offsetting, or extruding a large number of faces, etc., this can be worked around at the cost of increased memory use by creating a new output mesh instead, as done in our implementation of volumetric Catmull-Clark subdivision.

Additionally, the specialization for simplex meshes does not allow for non-homogeneous, co-dimensional simplicial complexes, i.e., d-simplex meshes with k-simplices with $k < d$ that are not part of at least one $(k+1)$-simplex. The general mesh data structure does not have any limitations of this kind. Therefore, it can be used in such cases. When working with meshes of this type frequently, the general mesh data structure could be specialized to support implicit row offsets for all top-down operators, as done in the simplex mesh specialization.

4. Simulation System Matrix Assembly

Figure 4.1.: Left: outer surface of the high-resolution mesh with 1.7 million tetrahedra used in the evaluation of our method. Right: cut through a lower-resolution model to show its inner structure. The models are based on the Airbus FCRC bracket, a titanium 3D-printed part developed using simulation and topological optimization (see, e.g., [Kra17]).

This chapter is based on the following publication:

[MS18] **Mueller-Roemer, J. S.** and A. Stork.
"GPU-based Polynomial Finite Element Matrix Assembly for Simplex Meshes."
In: *Computer Graphics Forum* 37(7) (Pacific Graphics 2018), pp. 443–454.
DOI: 10.1111/cgf.13581.

Large parts of the publication are quoted verbatim with minor changes, extensions, and corrections.

4.1. Introduction

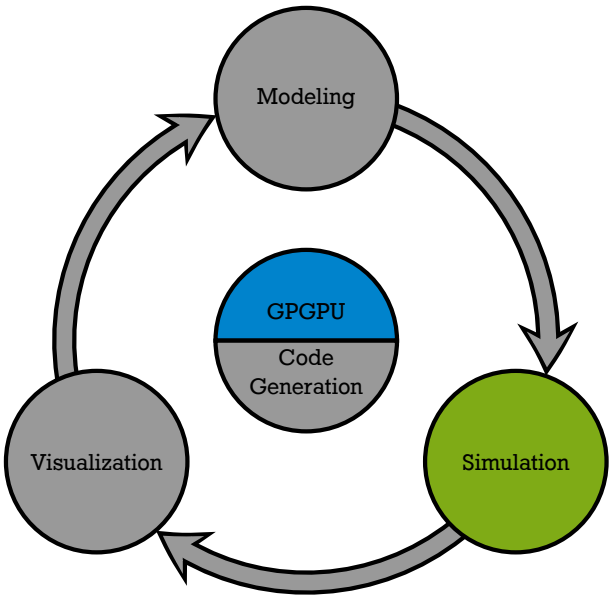


Figure 4.2.: Schematic representation of the virtual prototyping cycle, highlighting the approach used to accelerate the simulation step in this chapter.

In this chapter, we examine how the mesh data structures introduced in the previous chapter and the graphics processing unit (GPU) can be used to accelerate the simulation step of the virtual prototyping cycle shown in Fig. 4.2. Besides the use in topological optimization for 3D-printed parts, such as the geometry described by Kranz [Kra17] shown in Fig. 4.1, the continuously high cost of simulation for engineering in general but also for physically based animation in computer graphics makes efficient simulation methods more important than ever.

One of the most important simulation methods is the finite element method (FEM). Due to the ubiquity of FEM simulations, the constant need for faster simulations, and the price-performance benefits of GPUs, general purpose computing on the GPU (GPGPU) has been applied to them early on (see, e.g., a paper by Liu et al. a year after the first release of CUDA [LJWD08]). However, the main focus of research on GPU-based FEM has been the efficient solution of the resulting system of equations, not on the assembly of the sparse system matrix, which has come into focus as a significant bottleneck in high performance computing (HPC).

For example, a paper by Guo et al. states that assembly takes 30–40% of total computation time in their use cases [GLG+15]. This overhead is only amplified when a GPU solver is used with a CPU-based assembly approach. Therefore, we examine how the matrix assembly process can be accelerated on the GPU. Before the application of the improvements demonstrated in this dissertation, our solver, described in previous publications [WBS+13; WMA+15], took a median time of 14.1 s to set up the initial sparse matrix for the mesh shown on the left in Fig. 4.1, while solving the corresponding static simulation to convergence only takes 1.5 s. Additionally, mesh loading and CPU-based pre-processing took 15 s and could be accelerated by leveraging the GPU as described in Chapter 3.

The system matrix assembly process can be further split into two components:

1. Determination of the sparsity pattern of the system matrix.
2. Summation of the element stiffness matrices into the global system matrix.

During initial setup of a simulation or whenever the mesh changes, for example due to an adaptive method such as the adaptive FEM cloth simulation by Bender and Deul [BD13] or the adaptive, FEM-based brittle fracture simulation by Koschier et al. [KLB14], the sparsity pattern of the matrix must be determined. Cutting or tearing also result in changes to the sparsity pattern. When there are no changes to the topology, only the summation step must be repeated. Three approaches are compared in Sections 4.4.4 and 4.5.2. In

dynamic simulations using the co-rotational FEM (see, e.g., [MG04]), the summation must be performed once per time step due to the changes in the rotation matrices. For the nonlinear FEM, multiple updates to the matrix are required while solving the system. The first step is extremely costly when single, linear simulations are performed and when mesh topology is changed frequently, as is the case when simulating the additive manufacturing (AM) process itself (see, e.g., [LCA18]).

Tetrahedral mesh generation algorithms, such as the recently published unconditionally robust technique by Hu et al. [HZG+18], are more robust than current hexahedral or hex-dominant meshing methods. Furthermore, even when modeling using general polyhedral meshes, tetrahedral meshes can be derived directly by applying a specialized subdivision step, as shown by Altenhofen et al. [ASSF17]. Therefore, simulation meshes are most commonly triangular and tetrahedral meshes, or more generally simplex meshes.

Next to the element type, choice of element order is important. Linear elements suffer from shear and volume locking, which introduces artificial stiffness into the system (see, e.g., [ISF07]). Furthermore, increasing element order, i.e., the polynomial degree of their basis functions, can improve simulation quality at a lower cost than increasing mesh resolution (see, e.g., [WKS+11]). Therefore, we examine matrix assembly for the FEM of arbitrary element order.

To answer the second research question

2. **Can these GPU-optimized data structures be used to perform system matrix assembly for the FEM and other simulation methods more efficiently?** If yes, how can memory overhead be reduced while maintaining or improving performance?

we examine the following aspects:

1. Is it possible to improve GPU system matrix assembly performance and memory use by limiting the input to (higher-order) simplex meshes and making use of the topological properties of simplex meshes?

2. Can assembly be performed efficiently directly into a GPU-optimized sparse matrix data structure?

3. Which summation approach should be chosen for which problem?

In the following, Section 4.2 outlines the relevant related work. In Section 4.3, we describe the concepts behind our matrix assembly method. Section 4.4 provides details on the practical implementation of our approach in CUDA. We list the results of our comparisons with other techniques in Section 4.5. Lastly, Section 4.6 summarizes the chapter, and lists contributions and limitations.

4.2. Related Work

In this section, we outline related work on GPU-optimized sparse matrix formats, FEM system matrix assembly, and polynomial finite element methods. For a general background on sparse matrix data structures and GPGPU, refer to Chapter 2. For a description of the mesh data structure used in this chapter, refer to Chapter 3, Section 3.4.4.

4.2.1. GPU-optimized Sparse Matrices

Beyond the general-purpose sparse matrix data structures listed in Section 2.2, specialized matrix formats that are better suited for computation on GPUs are frequently used. Several such matrix structures have been developed in recent years, significantly improving the performance of sparse matrix-vector products (SpMVs) and other algorithms on GPUs. Additionally, significant advances have also been made in processing compressed sparse row (CSR) matrices on GPUs, such as the improved SpMV and general sparse matrix-matrix product (SpGEMM) algorithms by Liu and Schmidt [LS15] and Liu and Vinter [LV15], respectively. While their algorithms significantly improve CSR performance, even greater improvements can be achieved by using data structures more suitable for GPU computing.

Based on the ELLPACK-ITPACK format (ELL), Vázquez et al. developed ELLPACK-R [VOFG10], a variant of ELL better suited for GPU computation. ELLPACK-R takes ELL and augments it with a row length. Due to the padded column-major order storage of columns and values, load coalescing and good performance are achieved on a wide variety of matrices. However, the padding significantly increases space requirements, especially for matrices with significant variance in the number non-zeros per row.

The Bin-CSR data structure introduced by Weber et al. [WBS+13] also stores its off-diagonal values in a padded, column-major 2D array. However, the 2D arrays are defined per bin, a fixed size group of rows. They choose a bin size matching the number of threads in a half-warp, as coalescing in compute capability (CC) 1.x GPUs (see Section 2.1.2) was based on half-warps. This leads to significant savings in storage requirements for matrices with highly irregular non-zero column counts per row. The main diagonal is stored as a separate dense array due to the assumption that it typically does not have zero-valued entries in physical simulation. Anzt et al.'s SELL-P format additionally adds padding to ensure bins have a row length that is a multiple of t, where t is the number of threads within a block assigned to the same row [ATD14]. By assigning multiple threads to a row and performing local reductions within each block, further performance gains are possible for matrices with many nonzeros per row.

A number of other formats [DLM11; OSV11; KHW+12; ZGG+14] use a similar approach, but sort the rows by their number of nonzeros before padding. While this leads to lower divergence and storage costs, the SpMV transforms the output vector into a permuted basis. The cost of undoing the permutation can be offset in iterative methods by changing column indices accordingly and permuting the input once before the first iteration and the output once after the last iteration. However, this means that these formats cannot be used as a simple drop-in replacement and properties such as low bandwidth or dense blocks may be lost. Furthermore, such formats are less suitable for system matrix assembly, as row lengths have not been determined yet.

Further performance improvements can be achieved by exploiting a priori knowledge of the structure of given matrices, such as the dense $e \times e$ blocks in the FEM. Weber et al. also introduce Bin-BCSR, which assumes constant size $e \times e$ subblocks [WBS+13]. They only store the first column index of each e-column block. However, that index is still stored e times, once for each row that the block spans. Furthermore, while the diagonal is stored separately, the diagonal blocks are stored along with the remainder of the matrix with explicit zeros on the main diagonal. Similar approaches exist for sorted row formats with both fixed-size [CSV10] and varying-size [Ren12] blocks. While duplication of indices along rows is avoided in

these formats, they have the same disadvantage as the other sorted formats that they cannot be used as a drop-in replacement or for direct assembly.

4.2.2. System Matrix Assembly

Many works on GPU-based system matrix assembly focus only on the summation step. For example, there is the mesh-coloring-based approach by Komatitsch et al. [KME09]. The method by Weber et al. that uses a CSR-like encoding to store a mapping for each nonzero to the element stiffness matrices that influence it [WBS+13]. Assembly is performed directly into their GPU-optimized Bin-BCSR matrix data structure (see Section 4.2.1). Reguly and Giles compare several methods of summation into both CSR and ELL sparse matrices, and a matrix-free approach [RG15].

Cuvelier et al. describe a FEM matrix assembly method for vectorized languages [CJS15]. However, their focus is on the ease and efficiency of implementation in vectorized interpreted languages, such as Matlab or Python with numpy. Their CUDA implementation only achieves a speedup of 7× compared to the serial C implementation. One of the more comprehensive works, a paper by Cecka et al. [CLD10], compares five different assembly approaches that include determination of the sparsity pattern.

However, Zayer et al. have recently published two papers outlining their approach to GPU-based FEM stiffness matrix assembly that achieves even better performance [ZSS17a; ZSS17b]. Their method is based on the highly optimized GPU SpGEMM by Liu and Vinter [LV15]. To achieve high performance, a binning approach is used to choose between one of three different approaches to matrix assembly according to an estimated number of non-zero entries in a row. This leads to a comparatively complex implementation. Furthermore, as the number of non-zero entries is conservatively estimated, a temporary sparse matrix is allocated for assembly before being copied into the final sparse matrix. This leads to a large memory overhead.

In the domain of HPC, very large meshes are used that no longer fit into a single GPU's memory. Therefore the focus in HPC lies primarily on how to best perform the domain decomposition. Thébault et al. [TPD15] present a hybrid approach that combines domain decomposition, divide-and-conquer and mesh coloring for stiffness matrix assembly. Like Zayer et al., we focus on assembling stiffness matrices for small to medium size meshes that fit into GPU memory, as these are more common in simulation during early product design stages and optimization problems. However, combined with a domain decomposition approach, fast assembly on a single compute node is beneficial in HPC as well.

4.2.3. Polynomial FEM

In addition to the commonly known linear finite elements, elements with quadratic, cubic, or higher-order basis functions can be used (see [ZT00]). While linear elements have one node with e degrees of freedom per vertex, higher-order elements provide additional degrees of freedom. For quadratic basis functions, one additional node is added on every edge of the element and cubic elements provide two additional nodes per edge and one per face. As our matrix assembly method is not only targeted at linear finite elements but at arbitrary polynomial degree FEM, we discuss several important works here.

The choice of basis functions significantly affects the performance of the solver, as many basis functions require numeric integration to determine the element matrices. Lagrange polynomials are very commonly used and covered in most FEM literature, for example in Zienkiewicz and Taylors book [ZT00]. Similarly common are the "serendipity" functions, which only add nodes to the edges of the elements. These are also covered in the aforementioned book.

A wide array of other families of basis functions are used, such as the Legendre polynomials and non-uniform rational B-splines (NURBS) are used as well. Willberg et al. compared several of these polynomials for the use case of simulating Lamb waves [WDV+12].

In simulation for animation, the Bernstein-Bézier polynomials have been used, as the variational integrals can be computed in closed form for constant metric elements of any order, i.e., when undeformed positions are geometrically linear. For example, Weber et al. presented a method for interactive deformation simulation using Bernstein-Bézier polynomials and the co-rotational method [WKS+11]. Furthermore, they observe how the local topology in a tetrahedral mesh can inform the matrix assembly process. However, they do not apply this idea to matrix assembly on the GPU, nor do they generalize it to other element dimension (edge or triangle meshes for example) or other polynomial orders. Weber et al. later extended their analytic integration approach to cubic elements and used them for a p-multigrid finite element solver that employs a hierarchy not in mesh resolution, but in the polynomial degree p of the elements [WMA+15].

More recently, Feng et al. have demonstrated a curved optimal meshing procedure based on Bernstein-Bézier basis function [FAB+18]. Meshes with optimal, curved, variable metric tetrahedra are numerically advantageous compared to meshes with constant metric elements and can represent boundary geometry more accurately. However, closed form integration is no longer possible, potentially significantly slowing down element stiffness matrix computation.

4.3. Concept

In this section, we describe the theoretical concepts behind our matrix assembly method for higher-order simplex meshes. We determine the exact number of nodes that contribute to any individual node's row in the system matrix using only local topological information. No traversal of complex, pointer-based structures or sorting and discarding of duplicate non-zero indices is necessary.

First, let us briefly review how system matrices in the FEM are constructed. As an example, let $\mathbf{K} \in \mathbb{R}^{ne \times ne}$ be the stiffness matrix of a deformation simulation, where n is the number of nodes in the mesh, and e is the embedding or physical dimension, typically 3. The matrix $\mathbf{K}$ is sparse, but consists of locally dense $e \times e$ blocks. Therefore, we can also describe $\mathbf{K}$ as a fourth-order tensor $K_{ijkl} \in \mathbb{R}^{n \times n \times e \times e}$ that is sparse in ij. In this notation, we can describe the assembly process as

$$K_{ijkl} = \sum_{T \in \mathbb{T}_{ij}} E_{T,\mathcal{L}_T(i)\mathcal{L}_T(j)kl}, \tag{4.1}$$

where $\mathbb{T}_{ij} = \{T \mid i \in T \wedge j \in T\}$ is the set of all top-level elements, e.g., tetrahedra, that contain both nodes i and j. $\mathcal{L}_T$ is a map from global indices in $[1, n]$ to element-local indices in $[1, m]$, where m is the number of nodes in an element. Finally, $E_{T,ijkl} \in \mathbb{R}^{m \times m \times e \times e}$ is the dense element stiffness tensor of element T (or

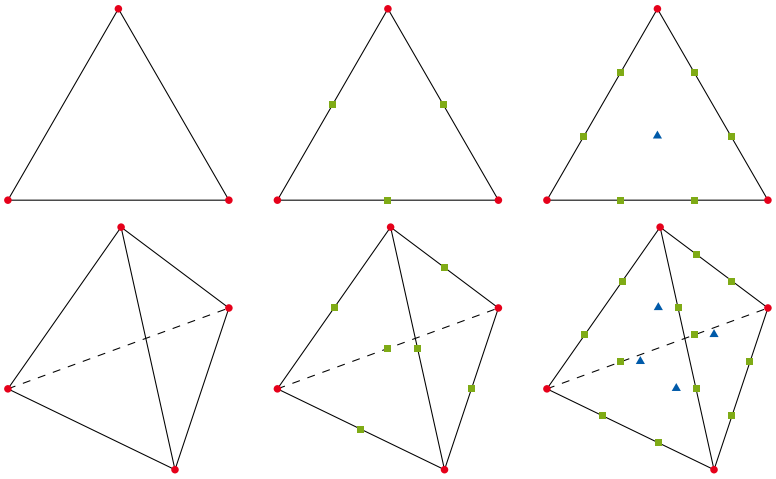


Figure 4.3.: Upper row: Tri3, Tri6, and Tri10 surface elements corresponding to $k = 2$ with $p = 1$, 2, or 3, respectively. Lower row: Tet4, Tet10, and Tet20 elements corresponding to $k = 3$ with $p = 1$, 2, or 3, respectively. Vertex nodes are shown as red circles, edge nodes as green squares, and face nodes as blue triangles.

element stiffness matrix $\mathbf{E}_T \in \mathbb{R}^{me \times me}$). We do not cover the construction of the element stiffness matrices $\mathbf{E}_T$, as it depends on many factors such as the choice of material model and basis functions, for example the commonly used Lagrange polynomials (see, e.g., [ZT00]) or Bernstein-Bézier polynomials (see, e.g., [WMA+15]), and is independent of the assembly process.

As mentioned in the introduction of this chapter, we limit our approach to simplex meshes to be able to determine the exact number of non-zero entries per row i. Simplices are the generalization of triangles and tetrahedra to an arbitrary number of dimensions $k \geq 0$. In this chapter, a simplex mesh is considered to be a homogeneous simplicial complex (see Section 3.4.4). This complex is embedded into an e-dimensional space, where $e \geq d$ and d is the topological element dimension.

As discussed in Section 3.4.4, every k-simplex has $k + 1$ vertices or nodes. For order-p elements, where $p \geq 1$, additional nodes are required. An order-p k-simplex has

$$\frac{(p+1)^{\overline{k}}}{k!} = \binom{p+k}{k} \tag{4.2}$$

nodes, where $x^{\overline{n}} = \prod_{k=0}^{n-1}(x+k)$ denotes the rising factorial. For varying p, these series of numbers of nodes correspond to the triangular or tetrahedral numbers, depending on k, shifted by one.

Figure 4.3 shows complete polynomial triangular and tetrahedral elements of orders one through three. We do not discuss incomplete higher-order elements, such as the cubic Tri9 or Tet16 elements that omit face points (see, e.g., [CGN18]), but adaptation of our method to such elements is straightforward. Furthermore, our focus is on continuous, conforming discretizations which share nodes between cells. In discontinuous Galerkin (dG) methods, nodes are unique to each top level element. Therefore, the resulting matrices have a simple, block-sparse structure and are generally easier to assemble. Assembly of dG system matrices is covered in the appendix of Di Pietro and Ern's book on dG methods [DE12].

As shown in Fig. 4.3, nodes are either only part of a top-level d-simplex element or one or more of its k-simplex subelements. We call nodes on the interior of a k-simplex k-simplex nodes, i.e., nodes on a

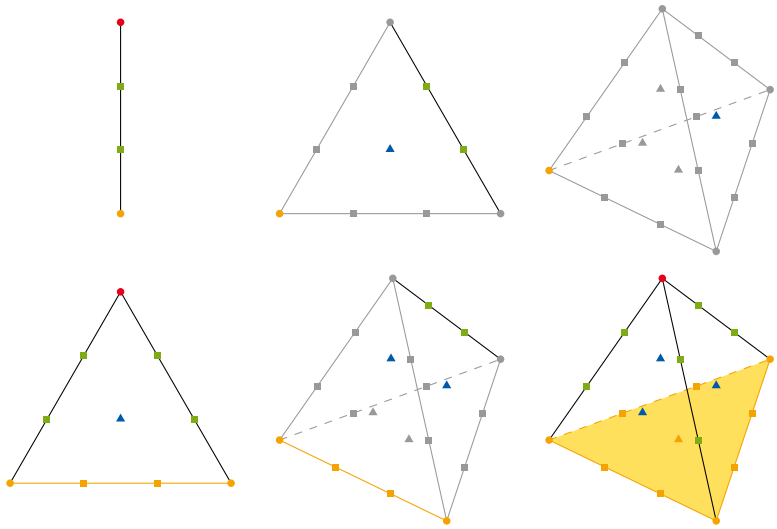


Figure 4.4.: From top-left to bottom-right: edge using a vertex, face using a vertex, cell using a vertex, face using an edge, cell using an edge, cell using a face. The current nodes, edges, and faces are marked in orange. Nodes and edges that are already part of other sub-simplices are grayed out. The nodes with unchanged color (red, green, or blue) are those that introduce new non-zero entries.

vertex are vertex nodes, nodes on an edge but not on a vertex are edge nodes, and so on. In the following equations, we define the binomial coefficient as

$$\binom{n}{k} = \begin{cases} \dfrac{n!}{k!(n-k)!} & \text{if } 0 \leq k \leq n \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

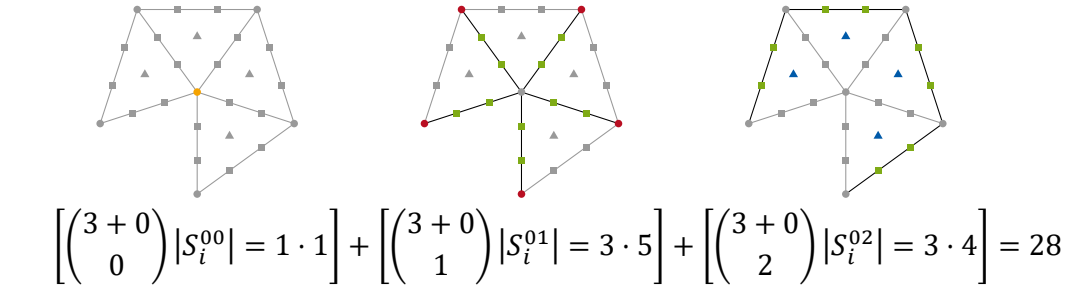
for notational simplicity. The number of k-simplex nodes on a k-simplex is

$$\frac{(p-k)^{\overline{k}}}{k!} = \binom{p-1}{k}. \tag{4.4}$$

This follows directly from the equivalence to the triangular or tetrahedral figurate numbers, as one "layer" of nodes is removed from all $k+1$ sides of the simplex, as can be seen in Fig. 4.3 when comparing the number of green edge nodes on an edge or the blue face nodes on a face to all nodes on that edge or faces.

To determine the number of other nodes that affect a node, we must examine the local neighborhood of each node. The number of non-zero entries for a node is the number of nodes in all top-level d-simplices (cells) that contain the node. However, we cannot simply multiply the result of Eq. (4.2) with the number of cells that use the node, as most vertices are part of more than one cell. This only provides an upper bound. For $d = 3$, vertex nodes are part of one 0-simplex, the vertex itself and sets of 1-simplices (edges), 2-simplices (faces), and 3-simplices (cells). Edge nodes are used by the corresponding edge, and sets of faces, and cells. Face nodes lie on the face, and a set of cells. Finally, cell nodes only belong to one cell. The six cases of elements used by other simplices are shown in Fig. 4.4.

To exactly determine the number of nodes that affect a k-simplex node, we use a bottom-up approach. First, we add the number of nodes on the originating k-simplex itself, which can be determined according

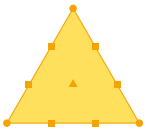


$$\left[\binom{3+0}{0}\left|S_i^{00}\right| = 1\cdot 1\right] + \left[\binom{3+0}{1}\left|S_i^{01}\right| = 3\cdot 5\right] + \left[\binom{3+0}{2}\left|S_i^{02}\right| = 3\cdot 4\right] = 28$$

(a) $k = 0$

$$\left[\binom{3+1}{1}\left|S_i^{11}\right| = 4\cdot 1\right] + \left[\binom{3+1}{2}\left|S_i^{12}\right| = 6\cdot 2\right] = 16$$

(b) $k = 1$



$$\left[\binom{3+2}{2}\left|S_i^{22}\right| = 10\cdot 1\right] = 10$$

(c) $k = 2$

Figure 4.5.: Example local neighborhoods of **(a)** a vertex ($k = 0$), **(b)** an edge ($k = 1$), and **(c)** a face ($k = 2$) in a third order ($p = 3$) triangular mesh ($d = 2$). The current k-facet i and all nodes on it are shown in orange in the leftmost images. Nodes that are drawn in color correspond to the current summand of Eq. (4.6) displayed below each image. For $k = d$, Eq. (4.6) only has one summand.

to Eq. (4.2). Then for every $(k+1)$-simplex that contains the k-simplex, we add the number of nodes on it, except for those nodes that are on the initial k-simplex. This continues with all $(k+2)$-simplices that contain the node, but now all nodes on $(k+1)$-simplices that use the node of choice must be ignored, as illustrated by the gray nodes in Fig. 4.4. As in the case of Eq. (4.4), the number of nodes added by each l-simplex with $l \geq k$ follows directly from the triangular/tetrahedral numbers, as exactly $l-k$ "layers" of nodes are removed, as can be seen in Fig. 4.4 by comparing the numbers of red, green and blue nodes to the total nodes on each element:

$$\frac{(p+1-(l-k))^{\overline{l}}}{l!} = \binom{p+k}{l}. \tag{4.5}$$

Using Eq. (4.5), the number of nodes affecting a k-simplex node i can be computed using

$$n_{k,i} = \sum_{l=k}^{d} \binom{p+k}{l} \left|S_i^{kl}\right|, \tag{4.6}$$

where S_i^{kl} is the set of l-simplices that are or contain the k-simplex of node i. This allows us to compute the number of nonzeros in any row of the matrix exactly using minimal topological information. In fact, only the sizes of the sets S_i^{kl} are required, not their contents. In other words, only the number of non-zero entries in the corresponding row of the (indirect) co-boundary operator $\left|S_i^{kl}\right| = \mathrm{nnz}_i\left(\hat{\mathbf{d}}_l^{l-k-1}\right)$ is required and not the column indices or values. Furthermore, the number is equal for all k-simplex nodes on a specific k-simplex. Figure 4.5 illustrates examples of the application of Eq. (4.6) for parts of a third order ($p = 3$) triangular mesh ($d = 2$).

4.4. Implementation

In this section, we explain how we used the concepts developed in Section 4.3 to implement a highly efficient GPU-based system matrix assembly method for arbitrary-order polynomial simplex meshes. Furthermore, we describe the extensions to the mesh representation introduced in Section 3.4.4 necessary for higher-order meshes and improvements to the GPU-optimized Bin-BCSR sparse matrix data structure.

4.4.1. Higher-Order Meshes

For the linear FEM, the only additional information required besides the topology stored in the data structure presented in Section 3.4.4 is the $n \times e$ array of positions. Although the embedding dimension $e = 3$ in our case, we do not store `positions` in transposed form for coalescing, as access patterns are mostly random which prevents coalescing. Therefore, we prefer to maintain locality instead. For higher-order simulations, we do not expand `cellVertices` to $n \times \binom{p+d}{d}$ entries as Zayer et al. do [ZSS17a; ZSS17b]. Instead we store the additional `cellEdges/edgeCell(Offset)s` and `cellFaces/faceCell(Offset)s` arrays of the corresponding (co-)boundary operators as required. As the number of k-simplex nodes on any k-simplex is constant and can be determined using Eq. (4.4), node indices are implicitly derived from the simplex indices. For $p < 3$ the storage requirements are equivalent to explicitly storing node indices per cell as in Zayer et al.'s approach. However, for $p \geq 3$ the storage requirements are smaller than explicitly storing all node indices, and independent of polynomial order. For example, the edges of the Tri10 and Tet20 elements shown in Fig. 4.3 have two edge nodes that can be stored as a single edge index instead of two node indices.

As we use constant metric polynomial tetrahedra, the undeformed positions of higher-order nodes can be implicitly derived from the vertex node positions using barycentric interpolation. Therefore, the size of the initial position array `positions` remains unchanged. Furthermore, the additional arrays listed above can be computed from `cellVertices` on demand instead of being loaded from a file (see Chapter 3).

To compute the numbers of non-zero entries in each row of the resulting matrix according to Eq. (4.6), further information is required. In particular, the number of edges per vertex is necessary for $p = 1$. For $p = 2$ the number of faces per vertex as well as the numbers of faces and cells per edge are required, and so on. Instead of computing the irregular, sparse matrices `vertexEdge(Offset)s` $\equiv \mathbf{d}_1$, `vertexFace(Offset)s` $\equiv \hat{\mathbf{d}}_2^1$, and `edgeFace(Offsets)s` $\equiv \mathbf{d}_2$, we only determine the regular 2D arrays `edgeVertices` $\equiv \boldsymbol{\partial}_1$, `faceVertices` $\equiv \hat{\boldsymbol{\partial}}_2^1$, and `faceEdges` $\equiv \boldsymbol{\partial}_2$. To determine the counts for Eq. (4.6), we use atomic additions of the corresponding binomial coefficient into the `columnCounts` array. This atomic count, without scaling by binomial coefficient, would be necessary for transposition in any case, as it is the first step of the `TCSRtoTCSC` method (see Listing 3.1), and we do not actually need to know which edges reference a vertex, for example.

4.4.2. Bin-BCSR*

As mentioned in the introduction, we aim to perform our matrix assembly directly into a GPU-optimized sparse matrix structure. As the Bin-BCSR matrix structure by Weber et al. [WBS+13] is also optimized for the FEM, we chose it as a basis for our implementation.

Listing 4.1: Dynamic scheduling for Bin-BCSR*. Bin size is chosen to match warp size. Before running, `counter` is initialized to 0.

```
auto tid = threadIdx.x;
auto lid = tid & (WARP_SIZE - 1);

auto bins = rows / WARP_SIZE + (rows % WARP_SIZE != 0);

int bin;
if(lid == 0)
    bin = atomicAdd(counter, 1);
bin = __shfl_sync(0xffffffff, bin, 0, WARP_SIZE);

while(bin < bins)
{
    auto idx = WARP_SIZE * bin + lid;

    if(idx < rows)
    {
        // parallel loop body
    }

    if(lid == 0)
        bin = atomicAdd(counter, 1);
    bin = __shfl_sync(0xffffffff, bin, 0, WARP_SIZE);
}
```

Bin-BCSR groups rows into bins with a constant and equal number of rows per bin. Within each bin, the maximum number of non-zero columns is determined. As in the ELLPACK-R format (see [VOFG10]), which does the same for all rows at once, other rows are padded with zeros to match that length and stored in transposed form to achieve coalescing. As done for each row in the commonly used CSR format, offsets are used to mark the position and the size of each bin. The blockwise dense nature of the matrix is exploited by omitting all but the first column index within a row of each dense $e \times e$ block.

However, we made several minor modifications that we collectively refer to as Bin-BCSR*:

1. Instead of only using the implied $e \times e$ block structure along columns, we use it along rows as well and store column indices referring to nodes instead of individual degrees of freedom. This simplifies the assembly, as column indices are not repeated for groups of e rows.

2. Consequently, we store all of the K_{ij00} entries in the block, followed by the K_{ij01} entries, etc. to maintain coalescing. As each thread now processes one i index of K_{ijkl} instead of an $e * i + k$ index into $\mathbf{K}$.

   One major advantage of processing rows of $e \times e$ blocks per thread is that fewer random loads are performed, as otherwise groups of e threads would load the same values.

   Consequently, the separately stored diagonal also stores an $e \times e$ block diagonal.

3. Instead of a bin size of 16 rows, we use a bin size of 32 rows, matching the size of a warp. This allows us to efficiently schedule warps dynamically as shown in Listing 4.1. This minimizes the effect of different length bins within a block.

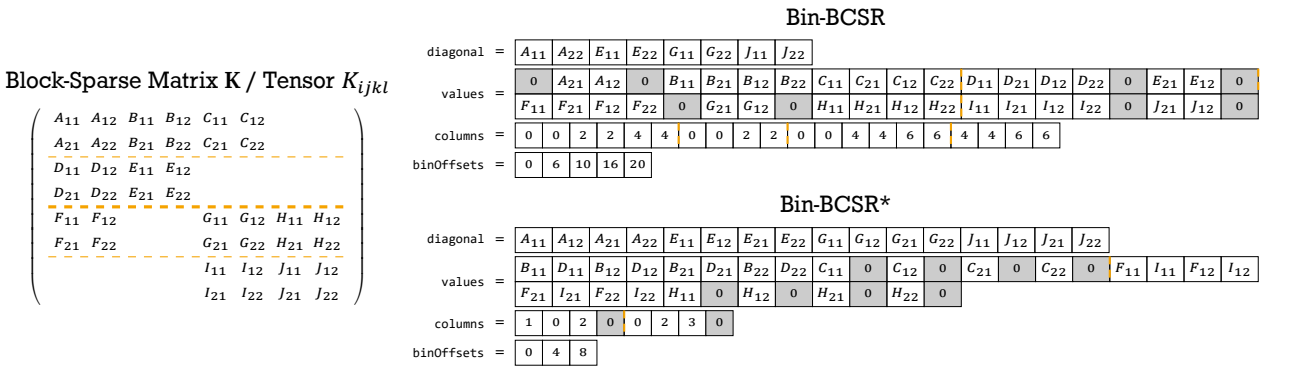


Figure 4.6.: Illustration of the differences between Bin-BCSR and Bin-BCSR* for an embedding dimension $e = 2$ / block size 2 × 2 and a bin size of 2. The dashed lines in the matrix mark which rows are combined into a bin with Bin-BCSR. The thick dashed line marks which rows are combined into a bin with Bin-BCSR*. The borders between bins are also indicated in the value and column arrays of the data structures. Padding and explicitly encoded 0-values are shown with a gray background. The column indices in the Bin-BCSR* structure are j-indices into the sparse tensor K_{ijkl}, while the column indices in the Bin-BCSR structure denote columns of the sparse matrix $\mathbf{K}$.

Examples of the original Bin-BCSR and improved Bin-BCSR* data structures are shown in Fig. 4.6. While the example only has padding in the Bin-BCSR* data structure as the bin size matches the block size for Bin-BCSR, the use of 16 row bins and 3 × 3 blocks in real-world use of Bin-BCSR also leads to a significant amount of padding. The increased amount of padding due to a larger number of rows per bin in Bin-BCSR* is mostly mitigated by the reduction in size of the column and offset arrays.

Due to the efficient SpMV possible with Bin-BCSR and Bin-BCSR*, these layouts are well-suited for use with iterative solvers such as the conjugate gradient method. Furthermore, use of the modified preconditioned conjugate gradient algorithm by Baraff and Witkin [BW98], or the improved version by Ascher and Boxerman [AB03], makes it possible to implement constraints without having to remove the corresponding lines from the matrix or otherwise modifying the sparsity pattern. The separate storage of the block diagonal in Bin-BCSR* allows for efficient implementation of (Block-)Jacobi preconditioners, which are highly parallelizable and therefore useful for GPU implementations (see, e.g., [ADFQ17]).

4.4.3. Sparsity Pattern

To determine the sparsity pattern, we first determine `columnCounts` as described in Section 4.4.1. We then determine the sizes of the bins by iterating over `columnCounts` in parallel and determining the maximum column count per bin, which can be implemented efficiently using warp shuffle instructions (instructions that exchange data between threads in a warp).

Afterwards, we compute the cumulative sum of these maxima (minus one for the diagonal and multiplied by bin/warp size) using a parallel prefix scan, another standard parallel primitive that is implemented in Thrust, CUB, and other libraries [BH15; Mer18]. These are the `binOffsets`. Finally, the total number of entries can be read back from the end of `binOffsets`.

This is then used to allocate the array of non-zero entry values (with $e \times e$ block entries) and column arrays. Additionally the main diagonal is allocated, but this can be done earlier or in parallel, completing the allocation of the final Bin-BCSR* matrix. The overhead with respect to a CSR or block compressed

sparse row (BSR) matrix due to binning is approximately 30–40% in our experiments, except for the smallest two meshes that incur a larger overhead.

After allocating the matrix structure, the column indices must be determined. We use a dynamically scheduled kernel based on Listing 4.1. For each row of the block matrix, we first determine which type of k-simplex the row/node belongs to, e.g., for $p = 2$ it is a vertex node if `idx` $<$ `numVertices` and an edge node otherwise. Depending on the node type, we iterate over `vertexCells` or `edgeCells` for that particular index. Every node in that cell (determined via the `cell*` arrays) is then added into a sorted array of column indices by first performing a binary search, then shifting back existing entries if a new value is inserted. Entries on the main diagonal are skipped due to the separate storage in Bin-BCSR*.

To do so efficiently, we reserve 63 indices per thread in shared memory for the column array. Each block has 64 threads or two warps, so this corresponds to 15.75 KiB of shared memory per block. While ≤ 63 columns are needed, all insertions are performed in shared memory. Upon insertion of the 64^{th} element or after all cells have been processed for that row, the columns are copied into the Bin-BCSR* structure. Further insertions, if any, are performed in global memory. In our experiments, only a negligible fraction of nodes had more than 63 neighbors.

4.4.4. Summation

Having determined the sparsity pattern, the last remaining step is the calculation and summation of the element stiffness matrices. At this point, the choice of material model, strain, and basis functions becomes relevant. Due to the simplicity of implementation, we use a linear, homogenous, isotropic material model, described using Lamé parameters and the stress-strain relationship

$$\sigma = 2\mu\epsilon + \lambda \text{tr}(\epsilon)\mathbf{I}, \tag{4.7}$$

where ϵ is the linear strain tensor, λ and μ are Lamé's first and second parameters, respectively, and σ is the resulting stress. Furthermore, we use the Bernstein-Bézier basis functions with constant metric tetrahedra due to the closed form integration. As shown by Weber et al. [WMA+15], computing the stiffness matrices becomes a simple weighted sum of outer products $\mathbf{b}_i\mathbf{b}_j^T$, where

$$\mathbf{b}_i = \left(\frac{\partial \xi_i}{\partial x}, \frac{\partial \xi_i}{\partial y}, \frac{\partial \xi_i}{\partial z}\right)^T \tag{4.8}$$

are the gradients of the barycentric coordinates ξ_i associated with the local vertex nodes i. While these choices allow for efficient, analytic integration, our method can be combined with other material models and basis functions requiring numerical integration as well.

We implemented three approaches to summation for comparison, called "per node", "inline", and "per entry" in the following. The first approach uses a *per node* map of nodes to cells, i.e., the same `*Cells` matrices used during pattern computation. The kernel is also very similar and uses the same shared memory cache, except all column indices are either loaded at the beginning if the row has ≤ 63 non-zero entries, or not at all. These loads are coalesced due to the transposed storage of column indices in Bin-BCSR*. For each node in each neighboring cell, a binary lookup is performed to find the correct position,

Table 4.1.: List of the sizes of the meshes used in the evaluation in numbers of vertices, edges, faces and cells. Additionally, the exact number of non-zero entries, the upper bounds resulting from Zayer et al.'s method, and the ratio between the two are given for the polynomial orders $p = 1$, 2, and 3.

Vertices	Edges	Faces	Cells	Order 1			Order 2			Order 3		
				Exact	Bound	Ratio	Exact	Bound	Ratio	Exact	Bound	Ratio
2.9 k	13.5 k	18.3 k	7.7 k	29.8 k	124.0 k	4.2	363.9 k	774.9 k	2.1	1.9 M	3.1 M	1.6
11.7 k	64.2 k	96.7 k	44.2 k	140.0 k	707.7 k	5.1	1.9 M	4.4 M	2.3	10.3 M	17.7 M	1.7
21.7 k	125.5 k	194.3 k	90.6 k	272.8 k	1.4 M	5.3	3.8 M	9.1 M	2.4	20.8 M	36.2 M	1.7
30.8 k	181.4 k	283.9 k	133.3 k	393.7 k	2.1 M	5.4	5.5 M	13.3 M	2.4	30.4 M	53.3 M	1.8
47.1 k	284.2 k	450.5 k	213.4 k	615.6 k	3.4 M	5.5	8.7 M	21.3 M	2.4	48.4 M	85.4 M	1.8
59.7 k	364.1 k	580.7 k	276.3 k	787.9 k	4.4 M	5.6	11.2 M	27.6 M	2.5	62.5 M	110.5 M	1.8
77.6 k	479.4 k	769.4 k	367.6 k	1.0 M	5.9 M	5.7	14.9 M	36.8 M	2.5	82.9 M	147.0 M	1.8
104.3 k	652.4 k	1.1 M	505.8 k	1.4 M	8.1 M	5.7	20.4 M	50.6 M	2.5	113.7 M	202.3 M	1.8
144.8 k	917.5 k	1.5 M	719.1 k	2.0 M	11.5 M	5.8	28.8 M	71.9 M	2.5	161.2 M	287.6 M	1.8
210.5 k	1.4 M	2.2 M	1.1 M	2.9 M	17.1 M	5.9	42.7 M	107.1 M	2.5	239.4 M	428.5 M	1.8
326.2 k	2.1 M	3.5 M	1.7 M	4.6 M	27.2 M	6.0	67.4 M	170.3 M	2.5	329.3 M	681.1 M	1.8

and the corresponding element stiffness matrix is loaded from global memory and added in global memory to the corresponding entry. The diagonal is calculated entirely in registers.

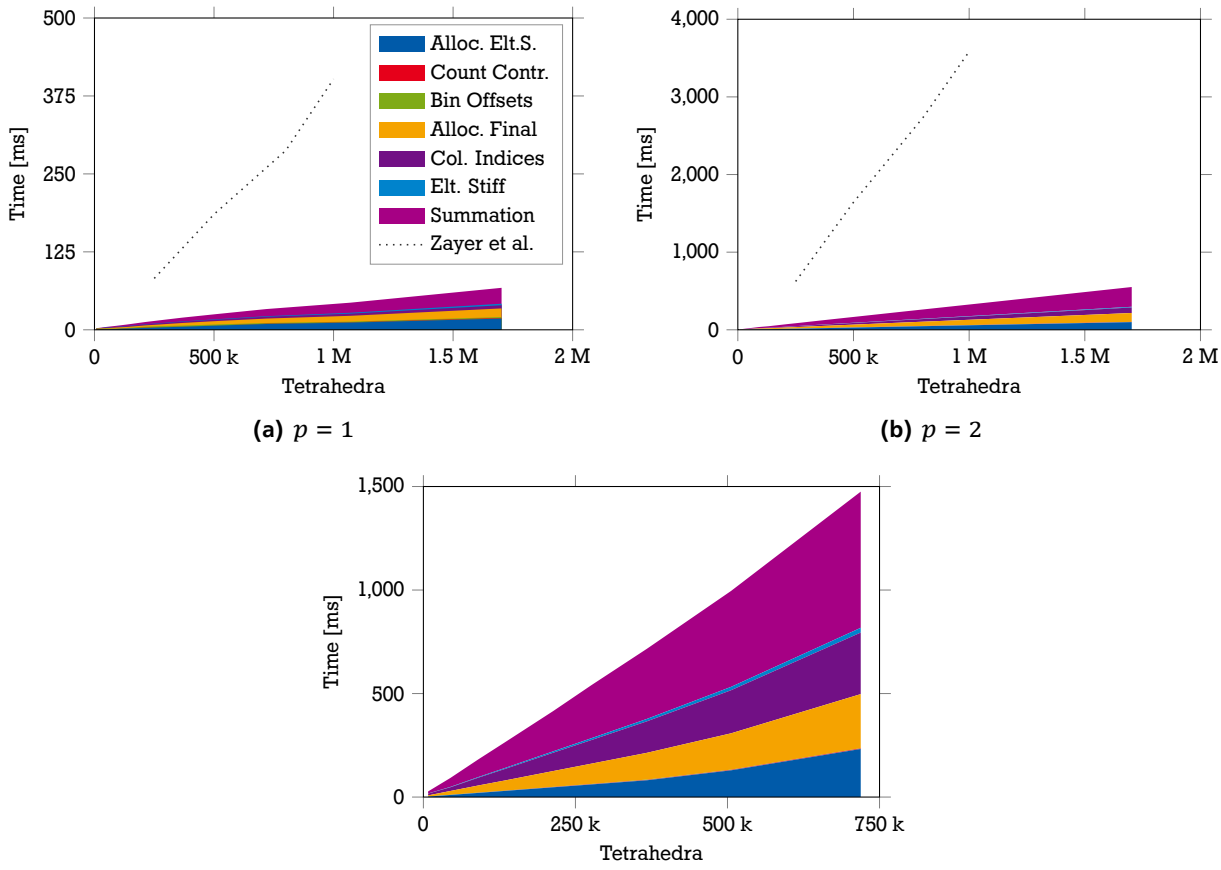
The *inline* approach is exactly the same, except that the element stiffness matrices are computed within the summation kernel. To reduce branch divergence and kernel size, the cell's vertex indices are reordered on the fly to move the current node to the beginning, while maintaining orientation by ensuring an even number of index swaps. As reordering becomes more complicated for higher orders, and register pressure was already very high, we only implemented this method for $p = 1$.

The final approach using *per entry* maps requires a large amount of additional data. We create an $n \times m$ sparse matrix with k non-zero column-only entries, where n is the number of non-zero entries in the system matrix, m is the number of $e \times e$ element stiffness sub-matrices and $k = \binom{p+d}{d}^2 \cdot |\mathbb{T}|$. This matrix stores the list of all element stiffness matrices that must be added for each entry. As each $e \times e$ entry can be processed in sequence, summation can be performed in registers, which is expected lead to performance improvements.

4.5. Results

Throughout all measurements for this chapter, we used a 64-bit Windows 10 computer with an Intel Core i7-6700K system processor (CPU) (4 cores, 4 GHz base clock), 16 GiB of DDR4-2133 main memory (34 GB/s) and an NVIDIA Quadro GP-100 GPU (3584 CUDA cores, 1.3 GHz base clock) with 16 GiB of HBM2 GPU memory (717 GB/s). All experiments were repeated 1000 times and results are given as the median value.

For the comparisons, we created 11 tetrahedral meshes with varying resolution of the model shown in Fig. 4.1. Table 4.1 lists the sizes of all meshes in vertices, edges, faces, and tetrahedra. Additionally, it lists the exact number of non-zero entries and the upper bound as calculated when using Zayer et al.'s approach based on Liu and Vinter's GPU SpGEMM [LV15; ZSS17a; ZSS17b]. Not only is the allocation of a temporary sparse matrix required, but it is 4.2–6.0× the size of the final matrix for $p = 1$ (an overhead of up to 600%), 2.1–2.5× the size for $p = 2$ (an overhead of up to 250%), and 1.6–1.8× the size for $p = 3$ (an overhead of up to 180%).



(a) $p = 1$ **(b)** $p = 2$ **(c)** $p = 3$

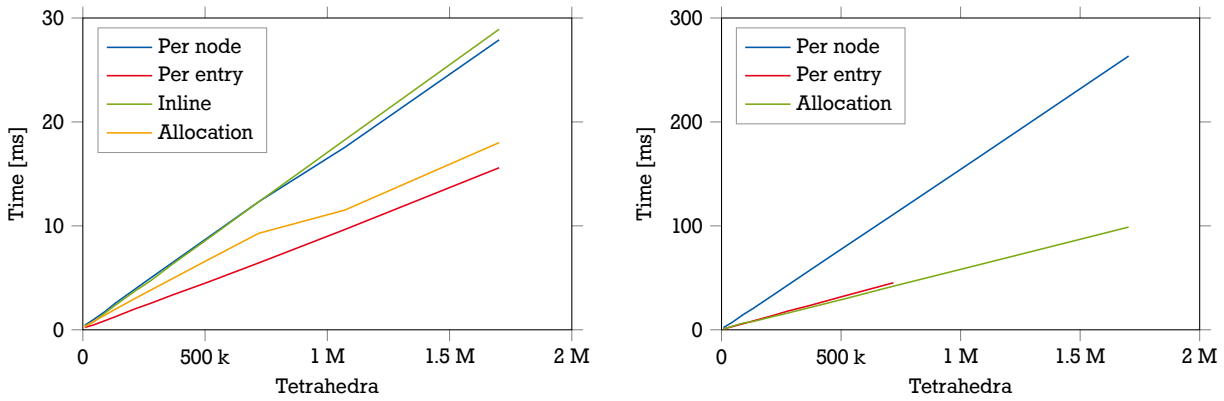
Figure 4.7.: Runtime of the entire assembly process over number of tetrahedra for $p = 1$, 2, and 3. Our method is shown split into the individual steps, where the colored areas represent the runtime of each step. The total height corresponds to the total runtime. The numbers for Zayer et al.'s method are taken from Figure 5 in their paper [ZSS17b].

4.5.1. Assembly

First, we compare the runtimes of the complete matrix assembly, including determination of the sparse pattern and element stiffness calculation and summation, with Zayer et al. For this comparison, we use the summation method with per node maps, as they do not require any information beyond the `vertexCells` and `edgeCells` sparse maps which are also required for the determination of column indices in each row. Furthermore, unlike the inlined method, which requires the same information, it is implemented for $p = 1$, 2, and 3. We discuss the three summation methods in detail in Section 4.5.2.

Figure 4.7 shows the times taken by the assembly process, including allocation and calculation (included in "Summation" which is discussed in more detail in Section 4.5.2) of the element stiffness matrices and the allocation of the resulting matrix, for $p = 1$, 2, and 3. The results of the individual measurements are also given in Table 4.2 at the end of this chapter. The numbers for Zayer et al.'s method are taken from Figure 5 of their paper [ZSS17b] and also given below:

Tetrahedra	250 k	500 k	800 k	1 M
Order 1	82.3	185	287	402
Order 2	623	1642	2732	3584



(a) $p = 1$ **(b)** $p = 2$

Figure 4.8.: Comparison of the runtimes of the three summation approaches for $p = 1$ and the two map-based approaches for $p = 2$. The methods with per node or per entry summation maps additionally require allocation of the element stiffness matrix array as a one-time cost (shown separately). For $p = 2$, the per entry map method failed for the two largest meshes due to allocation failures.

In particular, we used the timings given for SuiteSparse Sparse2 (see [Dav18]) in their paper, divided by the speedup given for their method on the GPU without reordering, as their numbers do not include the time taken for reordering or any other pre-processing steps.

A direct comparison is difficult, as Zayer et al. used a different set of hardware using an NVIDIA Tesla K40m (2880 CUDA cores, 745 MHz base clock) with 12 GiB of GDDR5 memory (288 GB/s). Therefore, we do not list any exact speedups. However, they use purely tetrahedral meshes for their evaluation as well, and the sizes of the meshes are on the same order of magnitude. The memory bandwidth of our GPU is 2.5× higher than the one used in their experiments. The theoretical peak single precision compute performance of the Quadro GP100 (10.3 TFLOPS) is similarly 2.4× larger than for the Tesla K40m (4.29 TFLOPS). Even when the larger factor of 2.5× is taken into account, a significant speedup of up to 3.9× for $p = 1$ and up to 4.4× for $p = 2$ remains.

Furthermore, the allocation of the final sparse matrix takes slightly more than 20% of the assembly time, both for $p = 1$ and 2, and only slightly less for $p = 3$. Since the allocation time is linear in the number of bytes allocated, as can be seen in the runtime measurements, introducing an additional 6×, 2.5×, or 1.8× oversized allocation for a temporary matrix would increase the runtime by more than 120%, 50%, or 32%, respectively, due to allocation alone.

In summary, by restricting our assembly method to simplex meshes, we were able to remove the 600%, 250%, or 180% memory overhead for $p = 1$, 2, or 3, respectively, caused by the temporary sparse matrix. Furthermore, we were able to achieve a significant speedup compared to the state of the art.

4.5.2. Summation

In Section 4.4.4, we described three different approaches to summation that differ significantly in the amount of memory they require. Figure 4.8 compares the runtimes of the three summation approaches, including calculation of the element stiffness matrices. The measurements are also given in Table 4.3 at the end of this chapter.

The method using per entry maps is 1.8–2.1× faster for $p = 1$ and 2.4–2.6× faster for $p = 2$ than when using per node maps (`vertexCells` and `edgeCells`). However, it requires a significantly larger amount of memory and fails for the two largest meshes when $p = 2$. For $p = 3$, only the per node approach has been implemented and is therefore not included in this section. The kernel with inlined computation of the element stiffness matrices (and per node maps), which was only implemented for $p = 1$, requires the least amount of memory and does not require allocation of the $e^2 \cdot \frac{n \cdot (n+1)}{2} \cdot |\mathbb{T}|$ floats for the element stiffness matrices, where n is the number of nodes per d-simplex (tetrahedron), and $\mathbb{T}$ is the set of all top level d-simplices. However, it is slightly slower than the per node map implementation due to the high register pressure and increased compute load.

Which summation method should be chosen strongly depends on the use case. The method with per node maps is the easiest to implement, as it requires no pre-processing beyond what is required for assembly itself, making it a good choice for simulations with adaptive meshes. Furthermore, it separates assembly and material model completely and makes implementation of complex material models possible.

The method with inline computation of the element stiffness matrices requires the same per node maps. However, even though many memory reads are avoided, it is slightly slower than the per node method. Furthermore, a very large number of registers are required, even for homogeneous, isotropic materials and linear basis functions. For more complex material models or higher-order basis functions, register spilling is effectively unavoidable, leading to even slower calculations. Despite these limitations, using the inline method may be worthwhile for static, linear simulations, as no time is needed to allocate the element stiffness matrix arrays, a smaller amount of memory is required, and the system is typically only assembled once.

Finally, the per entry map approach is up to 2.6× faster than the approach with per node maps. Like the per node method, it separates assembly and material model completely. Therefore, it should be the first choice for dynamic simulations using co-rotational strains as well as static or dynamic simulations using nonlinear material models or nonlinear strains, since both require frequent updates of the system matrix without changing the sparsity pattern. The large speedup is to be expected, as summation of the $e \times e$ blocks of the matrix can be performed in registers, unlike the other methods where summation takes place in memory. However, the memory overhead is significant and additional pre-processing is required.

4.6. Summary

In conclusion, we have shown that by restricting our method to simplex meshes (triangular and tetrahedral meshes), we are able to derive combinatorial equations that allow for exact allocation of the resulting sparse matrix using minimal topological information.

This allows us to perform assembly directly into a GPU-optimized sparse matrix structure based on Bin-BCSR by Weber et al. [WBS+13], while avoiding the 180% to 600% memory overhead, depending on polynomial order, and allocation time for the temporary matrix inherent to the current state of the art approach by Zayer et al. [ZSS17a; ZSS17b]. Due to the reduced memory requirements, we can simulate significantly larger meshes.

Furthermore, we achieve a significant speedup of up to approximately 4.4× compared to the state of the art. At the same time, our approach is easier to implement, as no sophisticated binning approach to choose between several algorithms, such as the one introduced by Liu and Vinter [LV15] and used by Zayer et al. [ZSS17a; ZSS17b], is necessary to achieve these speedups. Compared to the serial CPU-based assembly and mesh pre-processing previously used in our solver, speedups of up to 200× are achieved. This results in a theoretical total speedup of more than 18× for performing static simulations.

Additionally, we presented an improved version of Bin-BCSR, Bin-BCSR*, that uses the $e \times e$ block structure of the matrix along both dimensions to further reduce memory size and improve locality. By matching the bin size to the GPU's warp size, a simple, per-warp dynamic scheduling approach (see Section 4.4) can be used while processing the Bin-BCSR* matrix. This scheduling approach is used during matrix assembly and can be used to improve the speed of the SpMV as well.

Finally, we compared three approaches to the summation step that is required whenever material parameters change, on every frame in co-rotational FEM, or multiple times per step in non-linear FEM. We provide the reader with a basis for deciding which method to choose, depending on the use case, in Section 4.5.2.

This chapter answers the second research question posed in Chapter 1:

2. **Can these GPU-optimized data structures be used to perform system matrix assembly for the FEM and other simulation methods more efficiently?** If yes, how can memory overhead be reduced while maintaining or improving performance?

We have shown that system matrix assembly in the FEM can be accelerated significantly by combining efficient GPU-optimized mesh data structures as presented in Chapter 3 with an exact allocation approach that makes use of the topological properties of simplicial meshes and avoids overallocation completely. Furthermore, the approach is applicable to simplicial elements of arbitrary polynomial order. In the next chapter, we will explore code generation for sparse matrices with compound entries as an alternative to specialized formats such as Bin-BCSR and Bin-BCSR*.

However, as a direct result of the restriction to simplex meshes, mixed-element and hexahedral meshes cannot be used with the method presented in this chapter. For quadrilateral or hexahedral meshes, similar equations to the ones we presented can be derived. However, such meshes are frequently also structured. In such cases, the multidiagonal structure makes assembly trivial. For mixed-element meshes, a more general method, such as the one presented by Zayer et al. [ZSS17a; ZSS17b], should be used.

Furthermore, while our method should be well suited to adaptive meshes (h-adaptivity) due to its speed and memory efficiency, adaptive polynomial degrees per element (p-adaptivity) cannot be used. As polynomial degree can be different for every k-facet in p-adaptive approaches, any assembly method would have to store and retrieve the degree of all relevant facets instead of working with minimal topological information. An overview of h, p, and h-p FEM methods can be found in Babuška and Guo's paper [BG92].

Table 4.2.: Times shown in Fig. 4.7 for the individual stages of the assembly process with $p = 1$, 2, and 3, as well as the total time taken. All times are in milliseconds.

(a) $p = 1$

Tetrahedra	7.7 k	44.2 k	90.6 k	133.3 k	213.4 k	276.3 k	367.6 k	505.8 k	719.1 k	1.1 M	1.7 M
Alloc. Elt.S.	0.285	0.812	1.41	1.99	3.01	3.8	4.93	6.66	9.28	11.5	18
Count Contr.	0.644	0.475	0.0894	0.491	0.479	0.481	0.135	0.15	0.544	0.555	1.01
Bin Offsets	0.961	0.696	0.515	0.458	1.63	1.64	1.69	1.7	1.08	1.06	1.43
Alloc. Final	0.0169	0.466	1.62	1.57	1.94	2.68	4.19	5.41	7.1	9.32	13.8
Col. Indices	0.172	0.921	0.977	1.28	1.53	1.82	2.08	2.33	2.95	3.53	5.13
Elt. Stiff	0.0649	0.105	0.142	0.191	0.266	0.33	0.426	0.585	0.837	1.34	2.4
Summation	0.365	0.886	1.61	2.38	3.65	4.64	6.06	8.19	11.5	16.2	25.5
Total	2.51	4.36	6.36	8.36	12.5	15.4	19.5	25	33.3	43.5	67.3

(b) $p = 2$

Tetrahedra	7.7 k	44.2 k	90.6 k	133.3 k	213.4 k	276.3 k	367.6 k	505.8 k	719.1 k	1.1 M	1.7 M
Alloc. Elt.S.	0.772	3.36	6.49	8.04	12.6	16.1	21.3	29.3	42	62.2	98.8
Count Contr.	0.0731	0.12	0.151	0.162	0.925	1.02	1.01	1.44	1.78	2.32	3.44
Bin Offsets	0.0933	0.678	0.743	0.47	0.713	0.691	0.957	1.04	1.12	0.886	0.158
Alloc. Final	1.23	4.86	8.84	10.7	16.3	20.4	26.6	35.6	49.5	72.3	114
Col. Indices	1.51	2.51	4.81	6.01	9.39	11.9	16.2	21.7	30.6	45.2	71.3
Elt. Stiff	0.132	0.313	0.511	0.686	1.11	1.42	1.9	2.61	3.63	5.49	8.87
Summation	1.98	6.78	14.1	20	32	41.4	55.1	75.7	107	160	254
Total	5.79	18.6	35.6	46.1	73	93	123	167	236	348	551

(c) $p = 3$

Tetrahedra	7.7 k	44.2 k	90.6 k	133.3 k	213.4 k	276.3 k	367.6 k	505.8 k	719.1 k
Alloc. Elt.S.	2.45	10.2	20.2	29.6	47.3	60.9	81.1	129	232
Count Contr.	0.166	0.181	0.579	0.581	1.28	1.53	1.76	2.3	2.96
Bin Offsets	0.574	0.557	0.949	0.604	0.738	0.813	1.68	1.13	0.93
Alloc. Final	4.91	18.3	34.3	49	76.4	98.1	129	175	261
Col. Indices	8.82	17.9	37.8	55.3	88.5	114	152	208	297
Elt. Stiff	0.394	1.74	3.12	4.55	6.76	8.97	11.6	16.2	23
Summation	9.92	43.1	85.5	124	194	256	339	463	658
Total	27.2	92	183	263	415	541	716	996	1480

Table 4.3.: Runtime measurements of the three summation methods as well as the time for element stiffness matrix array allocation shown in Fig. 4.8. All times are given in milliseconds.

Tetrahedra	Order 1				Order 2		
	Simplex	Entry	Inline	Alloc	Simplex	Entry	Alloc
7.7 k	0.43	0.226	0.387	0.285	2.11	0.82	0.772
44.2 k	0.991	0.473	0.761	0.812	7.1	3	3.36
90.6 k	1.75	0.876	1.61	1.41	14.6	5.88	6.49
133.3 k	2.57	1.26	2.37	1.99	20.7	8.56	8.04
213.4 k	3.91	2.01	3.73	3.01	33.1	13.7	12.6
276.3 k	4.97	2.54	4.7	3.8	42.8	17.9	16.1
367.6 k	6.48	3.36	6.33	4.93	57	23.3	21.3
505.8 k	8.77	4.53	8.66	6.66	78.3	32	29.3
719.1 k	12.3	6.43	12.3	9.28	111	45.2	42
1.1 M	17.6	9.65	18.3	11.5	165	—	62.2
1.7 M	27.9	15.6	28.9	18	263	—	98.8

5. Sparse Matrix Layout Generation

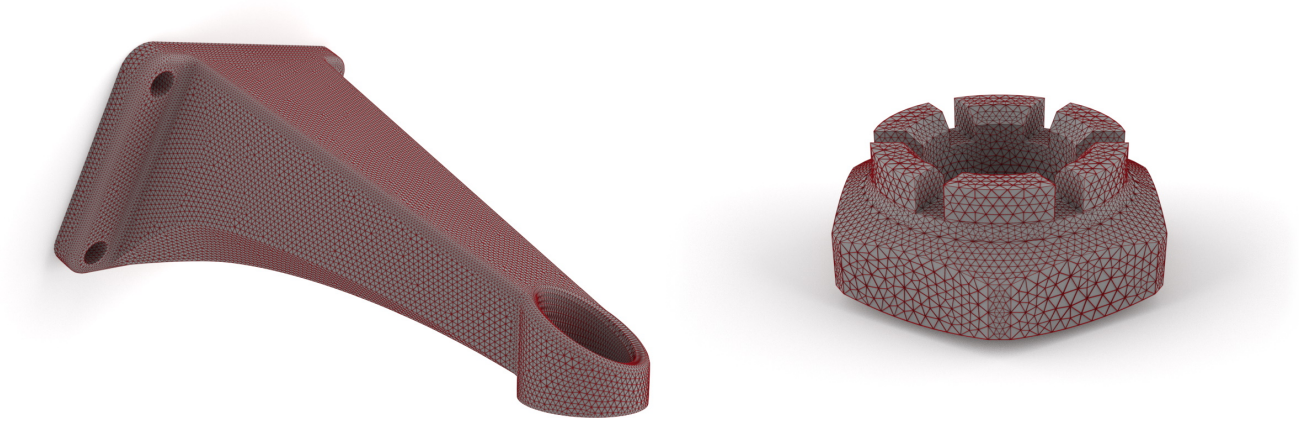


Figure 5.1.: Left: outer surface of the mesh corresponding to the matrix “fem/NX_MotorH67K”. Right: outer surface of the mesh corresponding to the matrix “fem/nut_37k”. Both meshes were created using a conventional CAD boundary representation (B-rep) and mesher, as illustrated in Chapter 3, Fig. 3.1. The matrices were assembled using the fast assembly method introduced in Chapter 4.

This chapter is based on the following publication:

[MSF19] **Mueller-Roemer, J. S.**, A. Stork, and D. W. Fellner. “Joint Schedule and Layout Autotuning for Sparse Matrices with Compound Entries on GPUs.”
In: *Vision, Modeling and Visualization*. VMV ’19. 2019, pp. 109–116.
DOI: 10.2312/vmv.20191324.

Large parts of the publication are quoted verbatim with minor changes, extensions, and corrections.

5.1. Introduction

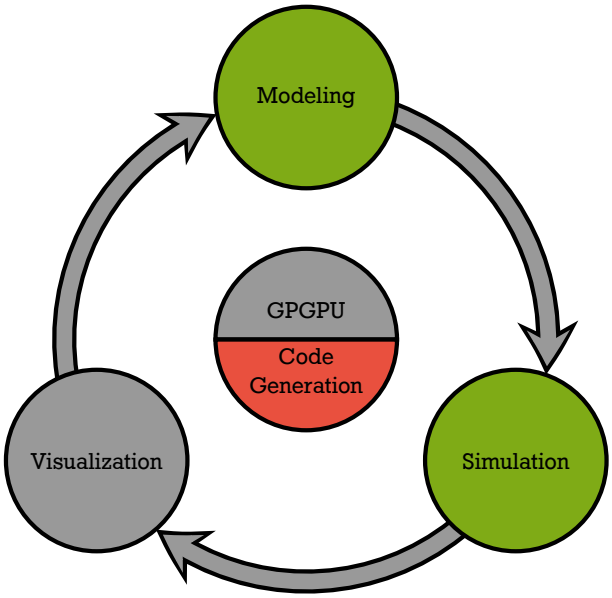


Figure 5.2.: Schematic representation of the virtual prototyping cycle, highlighting the approach used in this chapter to accelerate the modeling and simulation steps.

In this chapter, we examine how to accelerate the sparse matrix operations used in the modeling/mesh processing and simulation steps of the computer-aided engineering (CAE) cycle, as illustrated in Fig. 5.2. While the use of sparse matrices with complex coefficients in $\mathbb{C}$ is common in computational physics due to their ability to represent amplitude and phase in frequency-domain simulations, and therefore widely supported by linear algebra libraries, other extended number systems and compound entries are used in many areas of simulation, geometry processing, computer graphics, and computer vision. For example, the quaternions $\mathbb{H}$ have a long history of use in robotics and computer graphics due to their usefulness in representing and interpolating orientations in the special orthogonal group $\mathbf{SO}(3)$ (see, e.g., [Sho85]). More recently, their dual extension, the dual quaternion algebra, has seen increasing use for interpolation and averaging of rigid transformations in the special Euclidean group $\mathbf{SE}(3)$ as well (see, e.g., [KCŽO07]). Therefore, sparse matrices with quaternionic entries have found uses in fields such as the simulation of rigid multi-body systems (e.g., [Tas01]) or geometry processing (e.g., [CKPS18]). While workarounds using standard, real-valued matrices are available, these are typically inefficient in both performance and memory use (see Section 5.2.3).

In addition to extended number systems, the system matrices resulting from the finite element method (FEM) and other discretizations used in simulation and physically based animation of deformables exhibit dense 3×3-blocks (or more generally $e \times e$-blocks, see Chapter 4). Therefore, these matrices can be viewed as sparse matrices or tensors with 3×3-matrices as entries that are used with vectors of 3-dimensional vectors. We use the term *compound entries* as a generalization for both scenarios.

To make efficient use of the available hardware, especially manycore graphics processing units (GPUs), both memory layout and parallel schedule have to be chosen well. For example, interleaving the components of a compound entry leads to suboptimal performance on GPUs due to lack of coalescing (see Fig. 5.6 and Chapter 2). However, interleaving corresponds to how aggregate types, i.e., `struct` in C, C++, or CUDA, are defined in most programming languages. Depending on the specific hardware as well as the domain- and discretization-dependent distribution of non-zero entries, different parallel schedules, e.g., dynamic or static scheduling, and different block sizes are necessary to achieve good performance.

In this chapter, we examine how the concept of layout optimization used in dense array autotuners such as MATOG [WG17] can be applied to sparse matrices with compound entries on the GPU. Furthermore, we work towards tuning and generalizing over sparse matrix formats such as compressed sparse row (CSR),

ELLPACK-R [VOFG10], and Sliced ELLPACK (without reordering) [MLA10]. Additionally, our autotuner performs schedule optimization to deal with matrices with varying nonzero patterns and GPUs with a varying number of cores.

To answer our third research question

3. **Can code generation and compiler techniques be used to efficiently implement GPU sparse matrix formats and algorithms required in simulation and mesh processing?** Specifically, how can the performance of sparse matrices with extended number systems (e.g., complex numbers and quaternions) and dense blocks be improved?

we focus on the following aspects:

1. Is it possible to improve GPU sparse matrix-vector product (SpMV) performance by performing memory layout optimization of sparse matrices with compound entries using code generation?
2. How large are the gains when performing joint schedule and layout optimization compared to schedule optimization alone?

In the following sections, we provide an overview of related work including sparse matrix formats and code generation, just-in-time (JIT) compilation and autotuning for GPUs, and use cases for sparse matrices with compound entries in Section 5.2. Furthermore, Section 5.2 describes workarounds for the lack of quaternionic matrix support in current linear algebra libraries. We detail our approach in Section 5.3, followed by listing the results of our evaluation in Section 5.4. Finally, we summarize the chapter and suggest avenues for future research in Section 5.5.

5.2. Related Work

In this section, we list use cases and outline related work on formats for sparse matrices with compound entries. Furthermore, an overview of related JIT compilation, code generation, and autotuning approaches is given. Finally, we describe the workarounds used with existing linear algebra libraries when dealing with quaternionic matrices. For a general background on sparse matrix data structures and general purpose computing on the GPU (GPGPU), refer to Chapter 2.

5.2.1. Sparse Matrices with Compound Entries

As mentioned in the introduction, sparse matrices with complex coefficients are relatively common in frequency-domain simulations such as acoustic (see, e.g., [Tho06]) and electromagnetic simulations (see, e.g., [Jin14]). It is worth noting that in the latter case, the results are often both complex and vector-valued. Therefore, the system matrices have dense 3×3-blocks of complex entries. As complex matrices are a common use case, commercial sparse linear algebra libraries, e.g., NVIDIA cuSPARSE [NVI18b], provide well-tuned algorithms operating on such matrices.

In the field of geometry processing, Crane et al. use sparse quaternionic matrices to compute conformal transformations of triangle meshes in $\mathbb{R}^3$ [CPS11]. Later publications based on the quaternionic Dirac operator defined by Crane et al. result in quaternionic matrices as well (see, e.g., [CPS13; LJC17; YDT+18]). More recently, Chern et al. use parallel transport of unit quaternions representing triangle orientations for

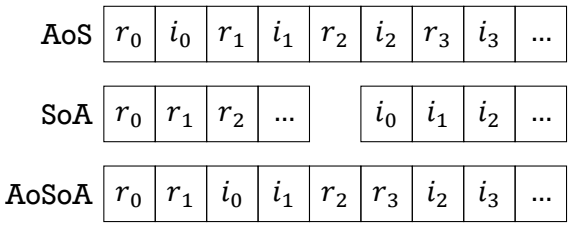


Figure 5.3.: Array of structures (AoS), structure of arrays (SoA), and array of structures of arrays (AoSoA) layouts of an array of complex numbers $c_k = r_k + i_k \mathbf{i}$. For AoSoA, an inner array size of 2 is shown.

the isometric immersion problem of orientable triangle meshes [CKPS18]. Both Crane and Chern et al. suggest using the 4×4-matrix expansion of quaternions, which leads to a significant memory and compute overhead (see Section 5.2.3).

In computer vision, Torsello et al. have applied dual quaternion graph diffusion to multi-view registration [TRA11]. Graph diffusion, like many other graph algorithms, can be expressed in terms of sparse linear algebra (see, e.g., [KAB+16]). While the resulting sparse matrix is real-valued, the vectors have dual quaternion entries.

In physically based animation, the commonly used FEM approach for simulating deformable models results in dense 3×3-matrix blocks. For this use case, libraries such as cuSPARSE support the block compressed sparse row (BSR) format, a variant of the CSR format (see, e.g., [Saa03]) for matrices with dense, fixed-size blocks that omits implicitly computable column indices. In academia, some researchers have used this fact to design GPU-optimized sparse matrix formats for FEM simulation. Examples include Weber et al.'s binned block compressed sparse row (Bin-BCSR) format [WBS+13], which only uses the block structure along one dimension and was improved in this dissertation to use it along both dimensions (see Section 4.4.2 or [MS18]).

For the simulation of flexible cables in interactive and virtual reality applications, Lang et al. introduce a quaternionic discretization of the rotational degrees of freedom of Cosserat rods [LLA11]. Furthermore, quaternionic matrices can be used to improve the performance of rigid multibody system simulations, as shown by Tasora [Tas01]. Despite their use in engineering, physics, and geometry processing, sparse quaternionic matrices are, to the best of our knowledge, not supported by any major (GPU-accelerated) linear algebra library.

5.2.2. Schedule and Layout Autotuning

The availability of the high-quality, liberally open sourced, optimizing compiler framework LLVM [LA04a] has lead to the development of several JIT-compilation approaches and domain specific languages (DSLs) for computer graphics and visualization. Examples range from the embedded DSL (eDSL) Halide for image processing [RBA+13] to compile time queries for remote visualization of simulation results [MA16]. Mullapudi et al. introduce bounds-analysis based heuristics for choosing parallel schedules for Halide image processing pipelines on system processors (CPUs) and GPUs [MAS+16]. Compared to naïve autotuning which involves a Cartesian product of scheduling options, their approach is far less costly.

While direct embedding of a compiler framework offers certain benefits such as being able to perform code generation and optimization without writing any files or starting external processes, it also significantly increases implementation complexity as considerations such as platform-dependent application binary interfaces (ABIs) become relevant. Alternatively, text templating techniques can be used along with standard compilers. In the domain of layout tuning of dense arrays for GPUs, Weber and Goesele have combined text template-based layout variations combined with model-based autotuning to great effect in MATOG [WG14; WG17]. Here, layout tuning refers to the selection of array of structures (AoS), structure of arrays (SoA), or array of structures of arrays (AoSoA) layouts (see Fig. 5.3) and row-major or column-major orders for dense n-dimensional arrays with compound entries.

In the area of compiler technologies for sparse matrices, Bik introduced a compiler that automatically transforms dense codes into sparse codes as well as performing CPU vectorization and parallelization [Bik96]. He also introduces more advanced transforms that require the sparsity pattern at compile time. In a more recent work, Cheshmi et al. also perform compile time analysis but combine it with a polyhedral loop optimizer to generate specialized direct solvers with improved vectorization on CPUs [CKSD17]. Venkat et al. use a runtime inspector and executer to analyze the sparsity pattern to perform reordering for GPU sparse matrix operations [VHS15]. They achieve speeds within $\pm 5\%$ of a hand-tuned implementation.

Kjolstad et al. introduce the tensor algebra compiler TACO which allows the user to select various layouts for each tensor in an tensor expression [KKC+17]. For two-dimensional tensors, the possible layouts correspond to dense matrices in row- or column-major layout, sparse CSR and compressed sparse column (CSC) formats, or their hypersparse (low-rank) extensions. However, the generated code is serial. In all cases, extended number systems are not supported.

Monakov et al. introduce the sliced ELLPACK format (see also Section 5.3.1) and perform tuning of slice and thread block size [MLA10]. Furthermore, they reorder matrix rows to achieve more compact storage. However, to efficiently use reordered matrices, permutation has to be performed rarely, e.g., before and after an iterative solver. Even though they do not consider compound entries and compare cards of the same generation, some cases are sped up by up to 10% when performing hardware-specific tuning.

5.2.3. Alternative Quaternion Representations

Quaternions, dual numbers, and dual quaternions, like complex numbers, belong to the Clifford algebras, which have equivalent, non-unique, real matrix representations (see, e.g., [HL90]). For example, quaternions can equivalently be represented as 4×4-matrices:

$$q = w + x\,\mathbf{i} + y\,\mathbf{j} + z\,\mathbf{k} \equiv \begin{pmatrix} w & -x & -y & -z \\ x & w & -z & y \\ y & z & w & -x \\ z & -y & x & w \end{pmatrix}, \tag{5.1}$$

where $q \in \mathbb{H}$, $w, x, y, z \in \mathbb{R}$, and $\mathbf{i}$, $\mathbf{j}$, $\mathbf{k}$ are the fundamental quaternion units. As a result, any quaternionic matrix $\mathbf{A} \in \mathbb{H}^{n \times m}$ can equivalently be represented as a real matrix $\mathbf{A}' \in \mathbb{R}^{4n \times 4m}$. However, this equivalence leads to a $4\times$ memory and compute overhead. For dual quaternions this overhead increases to $8\times$. The main advantage to this approach is that it allows the reuse of existing direct and iterative solvers.

Another approach is to decompose the matrices and vectors according to the Hamilton product:

$$\begin{aligned}\mathbf{A} &= \mathbf{W} + \mathbf{X}\,\mathbf{i} + \mathbf{Y}\,\mathbf{j} + \mathbf{Z}\,\mathbf{k}\\ \mathbf{q} &= \mathbf{w} + \mathbf{x}\,\mathbf{i} + \mathbf{y}\,\mathbf{j} + \mathbf{z}\,\mathbf{k}\\ \mathbf{Aq} &= (\mathbf{Ww} - \mathbf{Xx} - \mathbf{Yy} - \mathbf{Zz}) + (\mathbf{Wx} + \mathbf{Xw} + \mathbf{Yz} - \mathbf{Zy})\,\mathbf{i} +\\ &\quad (\mathbf{Wy} - \mathbf{Xz} + \mathbf{Yw} + \mathbf{Zx})\,\mathbf{j} + (\mathbf{Wz} + \mathbf{Xy} - \mathbf{Yx} + \mathbf{Zw})\,\mathbf{k},\end{aligned} \tag{5.2}$$

where $\mathbf{A} \in \mathbb{H}^{n\times m}$, $\mathbf{q} \in \mathbb{H}^m$, $\mathbf{W}$, $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z} \in \mathbb{R}^{n\times m}$, and $\mathbf{w}$, $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z} \in \mathbb{R}^m$. While this approach avoids the higher memory and compute overhead, it has other issues besides increased implementation complexity. First, the serial chaining of multiple matrix-vector products leads to increased latency, synchronization, and kernel launch overheads. Especially for small to medium-sized matrices, kernel launch overheads can make up a significant portion of execution time. Second, unlike the matrix expansion approach, most existing solvers cannot be used with this decomposition.

5.3. Concept and Implementation

In this section, we describe how we apply layout variations to sparse matrices with compound entries, the resulting code generator, and the autotuning approach.

5.3.1. Sparse Matrix Formats and Layouts

Much like layouts such as array of structures (AoS), structure of arrays (SoA), and array of structures of arrays (AoSoA) as well as row- or column-major orderings can be applied to dense n-dimensional arrays without changing the semantics of the array (see Fig. 5.3 and Section 5.2.2), we differentiate between semantically different sparse matrix data structures and those that only differ in their in-memory layout. For example, the CSR, ELLPACK(-R), and Sliced ELLPACK formats are all semantically arrays of length n of variable length arrays of tuples of column index and entry value for a matrix with n rows. Semantically different data structures such as the hierarchical data structure by Derler et al. [DZSS17] or the bitmap-based data structure by Zhang and Gruenwald [ZG18] are not considered to be layout variants.

CSR stores the column index and value tuples in a pair of contiguous arrays. Essentially, the tuples are stored in a 1D array in SoA layout. To mark the starting and ending positions of the per-row variable length arrays, CSR includes an array of $n + 1$ offsets into the contiguous arrays.

In the original ELLPACK format [RB85], the per-row arrays are padded with explicit zeros such that they all have the same length. The resulting dense 2D arrays are stored in column-major order. With respect to CSR, which is in row-major order by definition, the data is therefore padded and transposed. As the resulting arrays are dense, the offset array can be omitted. The ELLPACK-R format [VOFG10] replaces it with a nonzero count array of length n instead, which makes it possible to avoid performing any computations on added padding. If the stride between rows is additionally padded to a multiple of the warp size, the column-major ordering of these layouts leads to good coalescing on GPUs.

As ELLPACK and ELLPACK-R can lead to a very large memory overhead when a small number of rows has a much larger number of non-zero entries than the others, Sliced ELLPACK [MLA10] first partitions the

matrix into slices of k rows before padding and transposing the individual slices. As offsets within slices can be computed implicitly, only $\lceil n/k \rceil + 1$ offsets are required. For coalescing, k should typically be 16 (half-warp-sized) or 32 (warp-sized). The number of stored rows is padded to a multiple of k in the same way as the AoSoA layout requires padding the length of a dense array to a multiple of the inner array size.

We call the choice between row-major (CSR), padded column-major (ELLPACK-R), and sliced padded column-major (Sliced ELLPACK) orderings the *outer layout* of the sparse matrix. When compound entries are used, the dense entry array and the vector can be stored in AoS or SoA layouts. These choices define the *inner* and *vector layouts*.

While the CSC format of matrix $\mathbf{A}$ is identical to storing the transpose $\mathbf{A}^T$ in CSR format, we did not implement such transposed input layouts. As summation of each row and therefore entry of the output vector cannot be performed independently, supporting these requires different parallel algorithms. While Steinberger et al. have shown that the naïve approach of using atomic summation only leads to limited slowdown [SDZS16], doing so leads to the loss of determinism.

5.3.2. Code Generator

To generate the code for the layout variants, we use a text templating approach based on Jinja2, a templating language with Python and C++ implementations. The generated code is then compiled with the system C++ and CUDA compilers and linked as usual. In this chapter, we focus on the SpMV as it is the most costly component of iterative solvers such as the conjugate gradient algorithm.

To generate the code for a particular layout and schedule, the following inputs are required:

- A compound entry definition, i.e., a list of identifiers with associated types. Optionally, a separate definition can be given for vector entries.
- Multiplicative and additive operator definitions along with the additive neutral element (vector zero entry) and the matrix entry that results in it (matrix zero entry).
- The outer and inner layouts of the matrix, as well as the layout of the vector. If a sliced outer layout is chosen, the slice size must be defined as well.
- The schedule type (static or dynamic), as well as the numbers of streaming multiprocessors (SMs) n_s, blocks per SM n_b, and threads per block n_t.

The generated kernels use constant size blocks and grids, independent of the matrix size. Depending on schedule type, each block processes either chunks of n_t rows with a static stride of $n_s \cdot n_b \cdot n_t$ or selects chunks dynamically using an atomic counter. The block size n_t and the number of blocks per SM n_b are passed to the CUDA compiler using the `__launch_bounds__` annotation. This allows the compiler to generate code with the appropriate number of registers per thread.

As an additional performance optimization, the pointers to the vector array(s) for the right hand side are annotated with the `__restrict__` keyword. This allows the optimizer to use non-coherent loads which typically perform better for random access. Furthermore, AoS entries are annotated with the largest power

of two alignment between 1 and 16 of which their size is a multiple. This enables the use of vectorized loads where possible.

Besides the SpMV kernel, the code generator outputs header files for the generated matrix and vector classes. In addition to an interface callable from standard C++ code, the classes provide constructors to convert from CSR matrices in default AoS layout on the host to the chosen layout on the GPU. The source code of the generator is available for non-commercial use under https://github.com/fh-igd-iet/FhSparseGen.

5.3.3. Autotuner

Given a set of matrices as well as the entry and operator definitions, the autotuner jointly optimizes layout and schedule for the given matrices. To do so, it first determines the compute capability (CC) and the number of SMs n_s of the GPU. The compute capability, essentially the generation of the GPU, determines the warp size w (32 for all currently available NVIDIA GPUs), maximum numbers of blocks per SM, and threads per block supported by the GPU, as well as other indirectly relevant factors such as supported instruction set and number of registers per SM.

This information determines the bounds for the tuning parameters n_b and n_t. To limit the size of the resulting Cartesian product of variants, n_b is chosen from all 2^i and $3 \cdot 2^i$ with $i \geq 0$ that are within the bounds. Similarly, n_t is chosen from all $w \cdot 2^i$ and $w \cdot 3 \cdot 2^i$ that are within the bounds. The slice size k is limited to half-warp and warp sizes. The scheduling parameters (static/dynamic, n_b, and n_t) can either be tuned separately, or jointly with the layout parameters (outer, inner, and vector layouts).

The generated variants are built with the CUDA compiler, passing the compute capability as a parameter to generate code for the specific architecture. These are linked to a benchmarking fixture that calls and measures the runtime of the SpMV a given number of times for each matrix.

5.4. Results

In this section, we describe the setup of the benchmarks performed and evaluate their results.

Figures 5.4 and 5.5 show statistical information about numbers of non-zero entries and bandwidths of the rows of the matrices used in the evaluation. The extent of the boxes ranges from the lower to the upper quartile, with a line at the median. The whiskers extend from the minimum to the maximum. The bandwidth of a row is defined as

$$b_i = \max_{\left\{j \mid \mathbf{A}_{ij} \neq 0\right\}} j - \min_{\left\{j \mid \mathbf{A}_{ij} \neq 0\right\}} j \tag{5.3}$$

and provides information about the locality of accesses.

The matrices “mhd1280b”, “RFdevice”, “fem_filter”, and “mono_500Hz” are complex matrices from the SuiteSparse Matrix Collection [DH11] chosen to cover a large range of sizes and non-zero entry distribution patterns. The matrices beginning with “surface/” are quaternionic matrices that were generated from meshes available in the Stanford 3D Scanning Repository [Sta14] using Crane et al.’s algorithm [CPS11]. The matrices beginning with “fem/” are matrices with 3×3-block entries resulting from a lin-

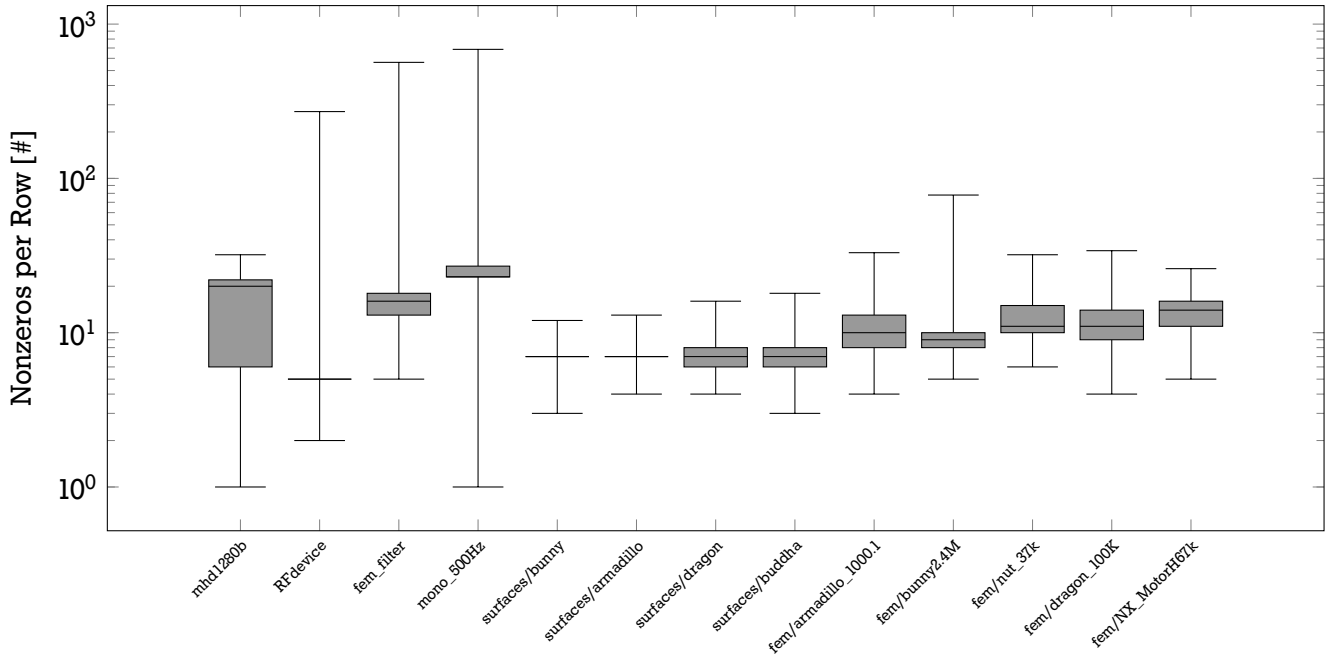


Figure 5.4.: Distribution of non-zero entries per row for each matrix used in the evaluation. For matrices beginning with "fem/", this is the number of non-zero 3 × 3-blocks per group of three rows.

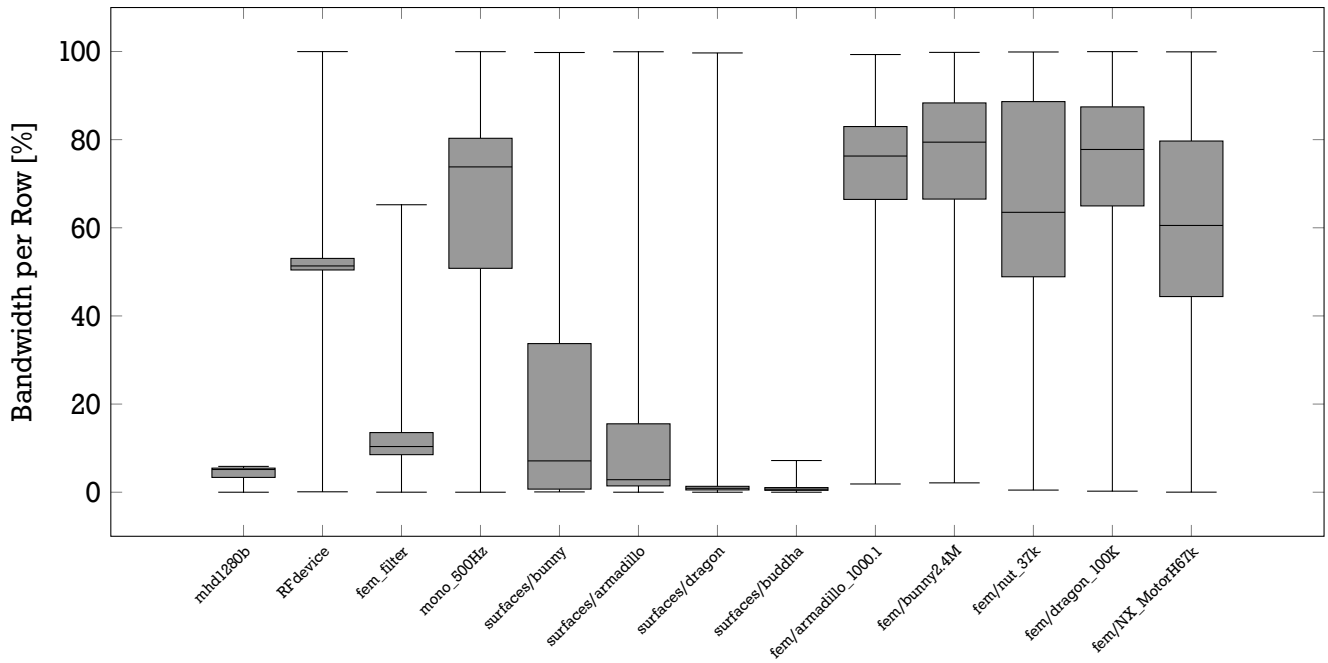


Figure 5.5.: Distribution of bandwidths per row for each matrix used in the evaluation. Given in percent normalized by number of rows/columns in each matrix. For matrices beginning with "fem/", block-row and -column indices are used.

ear FEM discretization on tetrahedral meshes and assembled using the methods introduced in Chapter 4, but converted to CSR and BSR for compatibility with existing libraries. The tetrahedral meshes were generated with Gmsh [GR09] (armadillo_1000.1), TetGen [Si15] (bunny2.4M, dragon_100K), CGAL [CGA18] (nut_37k), and Siemens NX [Sie18] (NX_MotorH67k). The meshes resulting in the latter two matrices are shown in Fig. 5.1. All matrices used in the evaluation are square.

The evaluations were performed on three machines with GPUs from various generations or CCs, including both professional and consumer (restricted double precision performance) GPUs, with the following hardware:

1. NVIDIA Quadro K2000 GPU (CC 3.0, 2 SMs, 2 GiB GDDR5), Intel i5-4670 CPU (4 cores, 3.4 GHz), 16 GiB DDR3-1600
2. NVIDIA GeForce GTX 980 GPU (CC 5.2, 16 SMs, 4 GiB GDDR5), Intel i7-4790K CPU (4 cores, 4.0 GHz), 16 GiB DDR3-1600
3. NVIDIA Quadro GP100 GPU (CC 6.0, 56 SMs, 16 GiB HBM2), Intel i7-6700K CPU (4 cores, 4.0 GHz), 32 GiB DDR4-2133

All systems were running Windows 10 and benchmarks were compiled with Visual Studio 2015 and CUDA 9.2.

To determine the best layout-schedule combination, the generated SpMV was called 1000 times per matrix for each combination. CUDA kernels were timed using CUDA events, to avoid primarily measuring the CPU-GPU synchronization overhead on small matrices.

All matrix-vector multiplications were also performed using cuSPARSE, NVIDIA's own highly tuned sparse linear algebra library. For the complex matrices, the built-in support for complex linear algebra was used. For the block-sparse matrices, the built-in support for the BSR format was used. For the quaternionic matrices, we used the matrix expansion (see Section 5.2.3) on the matrix only. $\mathbf{x} \in \mathbb{H}^n$ was represented as $\mathbf{x}' \in \mathbb{R}^{4n}$. As the resulting matrices are block-sparse too, BSR was used in this case as well. Therefore, only the number of values, not the numbers of offsets and column indices, of the matrix are quadrupled. The measured speedups are given in Figs. 5.6 and 5.7 for single and double precision, respectively.

5.4.1. Complex Matrices

The best layouts per GPU for each complex precision matrix as well as the speedups compared to only performing schedule tuning, i.e., using the "natural" CSR layout with entries in AoS layout, are given in Table 5.1. While most cases show an absolute speedup of less than 1× compared to cuSPARSE, speedups of approximately 1–1.5× are achieved for double precision matrices on the Quadro K2000 (see Figs. 5.6 and 5.7). Furthermore, the largest layout tuning gains are achieved on the K2000 as well. The speedups on the two newer GPUs are similar, despite the significantly lower double precision performance on consumer GPUs. While AoS is preferred for both inner and vector entry layout in most cases, no clear preference in outer layout can be observed. As both single and double precision complex entries can be aligned to 8 and 16 bytes, respectively, the preference of AoS layout is expected.

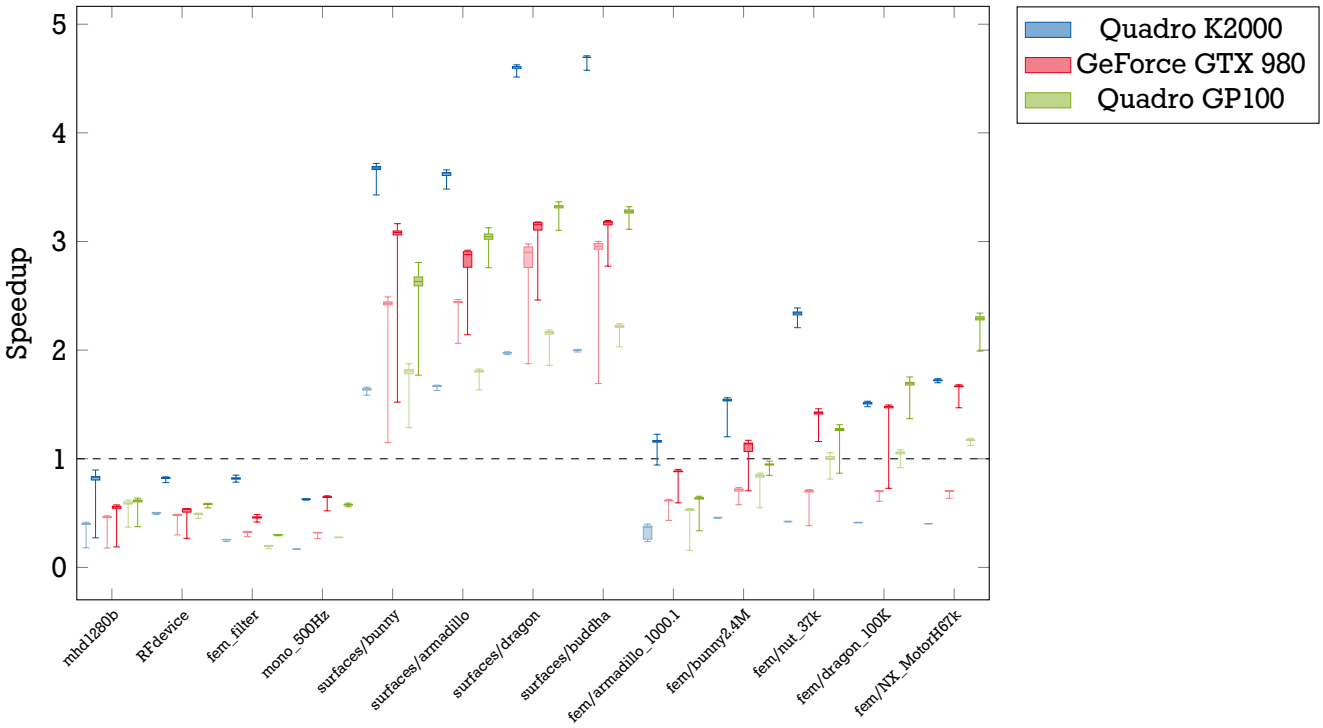


Figure 5.6.: Speedup relative to cuSPARSE in single precision with (dark) and without (light) layout optimization. For large compound entries ("fem/*") and extended number systems ("surface/*"), speedups of up to 4.7× are achieved.

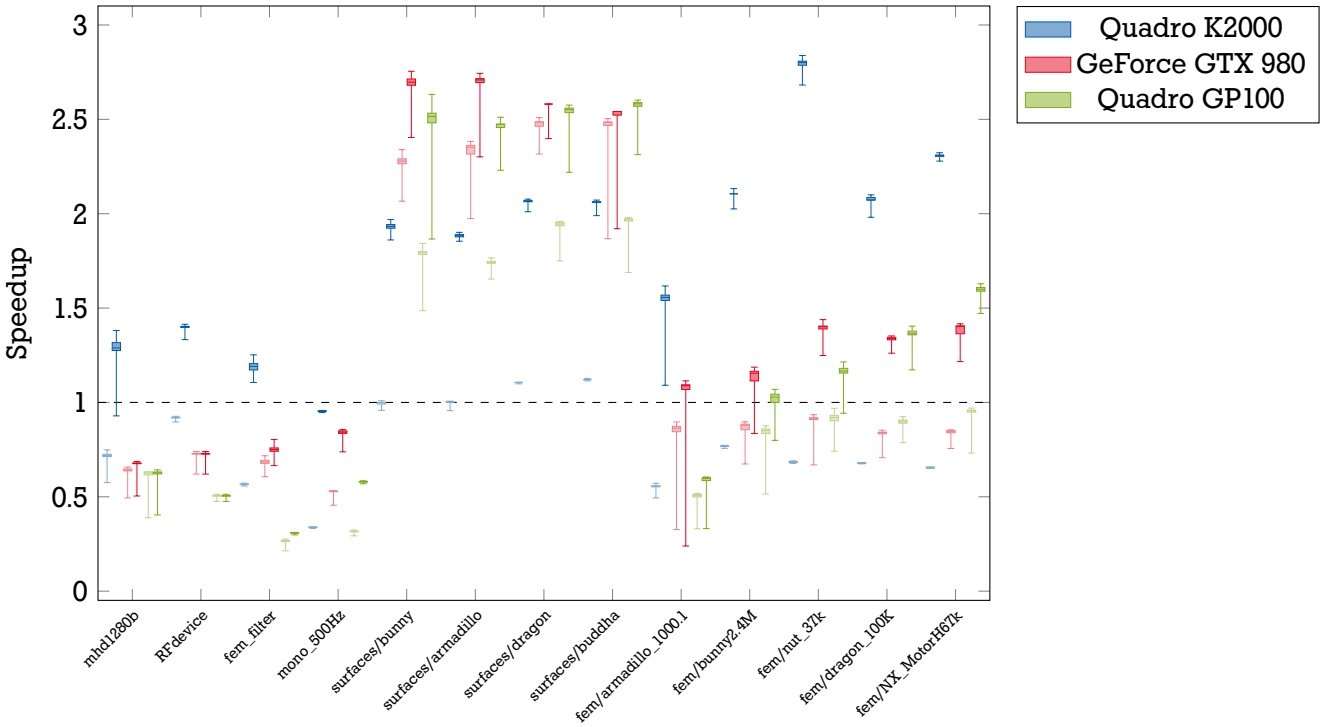


Figure 5.7.: Speedup relative to cuSPARSE in double precision with (dark) and without (light) layout optimization. For large compound entries ("fem/*") and extended number systems ("surface/*"), speedups of up to 2.8× are achieved.

Table 5.1.: Best layouts and layout tuning speedups for all complex single and double precision matrices. Layouts are given as outer-inner-vector, where ELL is ELLPACK-R and Sl-k is Sliced ELLPACK with a slice size of k.

Matrix	GPU	Single		Double	
		Layout	Speedup	Layout	Speedup
mhd1280b	K2000	ELL-AoS-AoS	2.09×	Sl-32-AoS-AoS	1.79×
	GTX 980	ELL-AoS-AoS	1.22×	Sl-16-AoS-AoS	1.05×
	GP100	CSR-SoA-AoS	1.05×	CSR-SoA-SoA	1.00×
RFdevice	K2000	ELL-AoS-AoS	1.64×	ELL-AoS-AoS	1.52×
	GTX 980	Sl-32-SoA-AoS	1.11×	CSR-AoS-AoS	1.00×
	GP100	CSR-SoA-AoS	1.18×	CSR-AoS-AoS	1.00×
fem_filter	K2000	ELL-AoS-AoS	3.20×	ELL-AoS-AoS	2.11×
	GTX 980	ELL-AoS-AoS	1.40×	ELL-AoS-AoS	1.10×
	GP100	ELL-AoS-AoS	1.52×	ELL-AoS-AoS	1.16×
mono_500Hz	K2000	Sl-32-SoA-AoS	3.74×	Sl-32-AoS-AoS	2.81×
	GTX 980	Sl-32-SoA-AoS	2.02×	Sl-16-AoS-AoS	1.59×
	GP100	ELL-AoS-AoS	2.09×	ELL-AoS-AoS	1.84×

5.4.2. Quaternionic Matrices

As for complex matrices in the previous section, we list the best layouts and speedups relative to not performing layout tuning for all quaternionic sparse matrices in Table 5.2.

Table 5.2.: Best layouts and layout tuning speedups for all quaternionic single and double precision matrices. Layouts are given as in Table 5.1.

Matrix	GPU	Single		Double	
		Layout	Speedup	Layout	Speedup
bunny	K2000	ELL-AoS-AoS	2.24×	ELL-SoA-AoS	1.93×
	GTX 980	ELL-AoS-AoS	1.27×	ELL-SoA-AoS	1.18×
	GP100	ELL-AoS-AoS	1.46×	ELL-AoS-AoS	1.40×
armadillo	K2000	ELL-AoS-AoS	2.18×	ELL-SoA-AoS	1.88×
	GTX 980	ELL-AoS-AoS	1.18×	ELL-AoS-AoS	1.15×
	GP100	ELL-AoS-AoS	1.68×	ELL-SoA-AoS	1.42×
dragon	K2000	ELL-AoS-AoS	2.33×	ELL-SoA-SoA	1.87×
	GTX 980	ELL-AoS-AoS	1.09×	ELL-SoA-AoS	1.04×
	GP100	ELL-AoS-AoS	1.54×	ELL-AoS-AoS	1.31×
buddha	K2000	ELL-AoS-AoS	2.35×	ELL-SoA-SoA	1.84×
	GTX 980	ELL-AoS-AoS	1.08×	ELL-AoS-AoS	1.02×
	GP100	ELL-AoS-AoS	1.48×	ELL-AoS-AoS	1.31×

While there was no clear outer layout preference in Section 5.4.1, the padded transpose (ELLPACK-R) is preferred in all cases. As can be seen in Fig. 5.4, the difference between the longest and shortest rows is much smaller for these matrices, therefore these matrices incur a significantly smaller amount of padding. As expected for the 16-byte aligned single precision quaternion entries, AoS layout is preferred in Table 5.2. Double precision quaternions are 32 bytes in size. Therefore, AoS layout requires two consecutive 16 byte loads and cannot achieve full coalescing. However, in all but two cases AoS layout continues to be preferred for the vector entries due to the mostly random access patterns. For the matrix entries, SoA is preferred in many but not all cases for the double precision matrices. As before, the largest speedups due to layout

tuning are achieved on the K2000. Unlike in the last section, the speedups on the GP100 are slightly larger than on the GTX 980, potentially due to the higher flops-per-byte ratio.

5.4.3. 3 × 3-block Matrices

As in the previous sections, we list the best layouts and speedups relative to not performing layout tuning for all sparse matrices with 3 × 3-block entries (and vectors of 3D vectors) in Table 5.3.

Table 5.3.: Best layouts and layout tuning speedups for all single and double precision matrices with 3 × 3 blocks. Layouts are given as in Table 5.1.

Matrix	GPU	Single		Double	
		Layout	Speedup	Layout	Speedup
armadillo_1000.1	K2000	Sl-32-SoA-SoA	3.15×	ELL-SoA-SoA	2.80×
	GTX 980	Sl-32-SoA-SoA	1.44×	Sl-32-SoA-SoA	1.26×
	GP100	Sl-16-SoA-SoA	1.20×	Sl-32-SoA-SoA	1.18×
bunny2.4M	K2000	ELL-SoA-AoS	3.34×	ELL-SoA-AoS	2.74×
	GTX 980	ELL-SoA-AoS	1.58×	ELL-SoA-AoS	1.31×
	GP100	ELL-AoS-AoS	1.12×	ELL-AoS-SoA	1.21×
nut_37k	K2000	ELL-SoA-SoA	5.54×	ELL-SoA-SoA	4.08×
	GTX 980	Sl-32-SoA-SoA	2.03×	ELL-SoA-SoA	1.53×
	GP100	Sl-32-SoA-SoA	1.27×	ELL-AoS-AoS	1.27×
dragon_100K	K2000	ELL-SoA-AoS	3.67×	ELL-SoA-AoS	3.05×
	GTX 980	ELL-SoA-AoS	2.12×	ELL-SoA-AoS	1.59×
	GP100	ELL-AoS-AoS	1.61×	ELL-AoS-AoS	1.52×
NX_MotorH67k	K2000	ELL-SoA-AoS	4.28×	ELL-SoA-AoS	3.52×
	GTX 980	ELL-SoA-AoS	2.37×	ELL-SoA-AoS	1.66×
	GP100	ELL-SoA-AoS	1.95×	ELL-SoA-AoS	1.67×

Both 3 × 3 blocks and 3D vectors cannot be aligned to power-of-two addresses without introducing padding, independent of scalar precision. Combined with the large entry size, the preference of SoA inner layout is expected. For the vector layout, AoS is preferred in most cases except for the fem/armadillo_1000.1 and fem/nut_37k matrices. As in the previous sections, the greatest gains are achieved on the K2000. This is followed by the GTX 980 and the GP100 benefits the least. Except for the smallest matrix, the tuned matrix layouts and schedules are faster than cuSPARSE using BSR as seen in Figs. 5.6 and 5.7.

5.5. Summary

In summary, we have shown that significant speedups can be achieved by performing joint schedule and layout autotuning for sparse matrices with compound entries. Compared to only performing schedule tuning, speedups of up to 5.5× are achieved (see Table 5.3). Compared to the highly tuned vendor library cuSPARSE, we achieve speedups of up to 4.7× for the SpMV (see Fig. 5.6). Even for matrices with dense blocks, which are supported directly in cuSPARSE, we achieve speedups of up to 2.8× using our approach (see Fig. 5.7). While the speedups are smaller than what can be achieved with sparsity pattern specific compilation approaches (see, e.g., [CKSD17]), similar matrices typically require similar layouts (see Table 5.2). Therefore, our approach can be applied to domain-specific tuning of SpMVs, which can

be performed beforehand for each new GPU using a domain-specific set of input matrices, resulting in shorter computation times, especially in computer graphics applications.

This chapter answers the third research question posed in Chapter 1:

3. **Can code generation and compiler techniques be used to efficiently implement GPU sparse matrix formats and algorithms required in simulation and mesh processing?** Specifically, how can the performance of sparse matrices with extended number systems (e.g., complex numbers and quaternions) and dense blocks be improved?

By generating GPU- and domain-specific matrix layouts and SpMV codes, computations on sparse matrices with extended number systems as well as block-sparse matrices can be accelerated significantly. Compared to only performing schedule optimization, joint schedule and layout optimization leads to significantly larger speedups, especially on older but also on current GPUs. In the following chapter, we will further explore code generation, but in the context of accelerating remote visualization.

While we have shown that significant speedups can be achieved by using joint schedule and layout tuning for the sparse matrix-vector product with compound entries, other procedures, e.g., dot products and matrix-matrix products, are necessary in many applications. Tuning the layouts for these procedures as well requires defining (domain-specific) weighting between the performances of the individual procedures, as changing layouts between operations would be costly.

For complex matrices, performance does not match the well-tuned operations provided by cuSPARSE, except on the older Quadro K2000 GPU. However, there is no reason not to use the well-supported vendor library in such cases.

We currently do not consider memory overhead due to padding in the tuning approach. Especially the padded transposed layout without slicing incurs large memory overheads. This would require weighting the performance and memory overhead for scoring. Alternatively, the compact CSR outer layout could always be generated as a fallback.

6. Streaming Post-Processing and Visualization

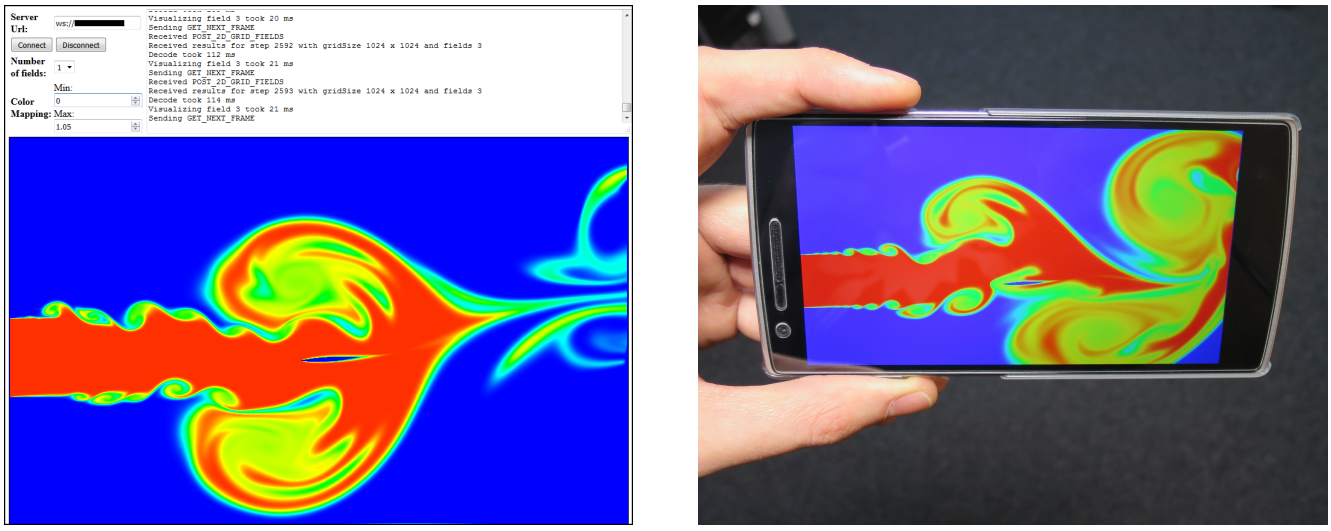


Figure 6.1.: The web-based streaming client can be run in a web browser on desktop computers or mobile devices without installing additional software and provides a simple user interface including 2D visualization and basic logging functionality.

This chapter is based on the following publications:

[WMSF15] Weber, D., **J. S. Mueller-Roemer**[1], A. Stork, and D. W. Fellner.
"A Cut-Cell Geometric Multigrid Poisson Solver for Fluid Simulation."
In: *Computer Graphics Forum* 34(2) (Eurographics 2015), pp. 481–491.
DOI: 10.1111/cgf.12577.

[MA16] **Mueller-Roemer, J. S.** and C. Altenhofen.
"JIT-compilation for Interactive Scientific Visualization."
In: *Short Papers Proceedings: 24th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*. WSCG '16. 2016, pp. 197–206.

[BGM19] Bormann, P., R. Gutbell, and **J. S. Mueller-Roemer**.
"Integrating Server-based Simulations into Web-based Geo-applications."
In: *Eurographics 2019 - Short Papers*. 2019. DOI: 10.2312/egs.20191012.

The bulk of the chapter is based on the second paper [MA16]. Section 6.3.2 is based on another publication on which I share primary authorship [WMSF15]. My contributions to that paper are the finite volume method (FVM) formulation leading to a consistent multigrid hierarchy, the OpenMP-parallelized system processor (CPU) implementation of the fluid simulation, and the graphics processing unit (GPU) implementation of the multigrid solver for the GPU-accelerated finite difference fluid simulation code with cut cells by Weber. Sections 6.3.5 and 6.4.4 are based on a short paper by Bormann et al. that I co-authored [BGM19]. My contributions to that paper are the improved rich pixel (rixel) encoding approach, improving on previous work by Altenhofen et al. [ADSF16] by significantly reducing required bandwidths, and the GPU-based implementation of the server side of and encoder for the hybrid renderer. Large parts of these publications are quoted verbatim with minor changes, corrections, and extensions.

[1]The two primary authors contributed equally to this work.

6.1. Introduction

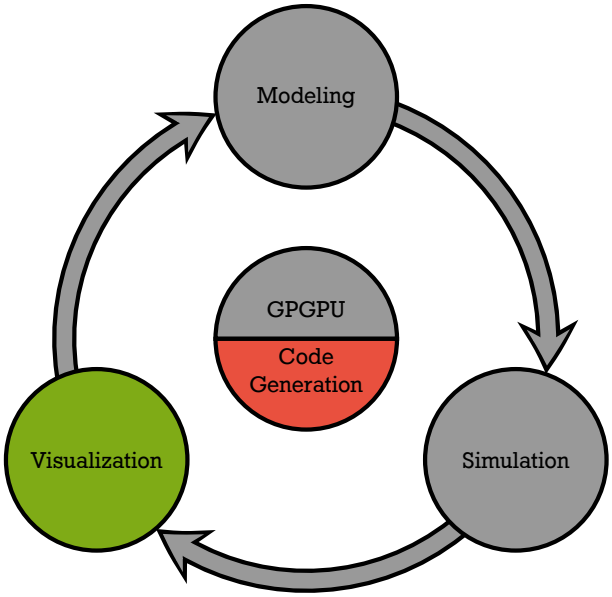


Figure 6.2.: Schematic representation of the virtual prototyping cycle, highlighting the approach used to accelerate remote visualization in this chapter.

In this chapter, we examine how code generation can be used to accelerate the visualization step of the computer-aided engineering (CAE) cycle shown in Fig. 6.2. In CAE workflows, compute-intensive simulations are more and more often run on remote cloud or high performance computing (HPC) infrastructures to improve availability by no longer requiring a high performance workstation or local HPC cluster. To avoid downloading large simulation results to a local client machine, solutions for remote post-processing and visualization are needed.

Although the option of using standard visualization tools via a video streaming system such as Virtual Network Computing (VNC) [RSWH98] is attractive, it is desirable to keep latencies to a minimum to increase usability [TAS06]. While modern video streaming solutions using hardware-based encoding can achieve very low latencies on local networks (see, e.g., [BMFG18]), any interaction incurs a latency of at least one round trip. By transferring (partial) floating point simulation data instead, operations such as probing or changes in color mapping can be performed locally with essentially zero latency. Similarly, by transferring geometry or point data in 3D, smooth camera interaction becomes possible [ADSF16].

When individual result fields of a simulation are visualized, data can simply be streamed from the server running the simulation. When viewing derived values that depend on multiple fields such as the total energy density $\frac{v^2}{2} + gz + \frac{p}{\rho}$ in an Eulerian computational fluid dynamics (CFD) simulation, a different solution is required, as the cost of transferring all data would be prohibitive, especially when considering comparatively slow mobile connections (see Fig. 6.3) and mobile power consumption.

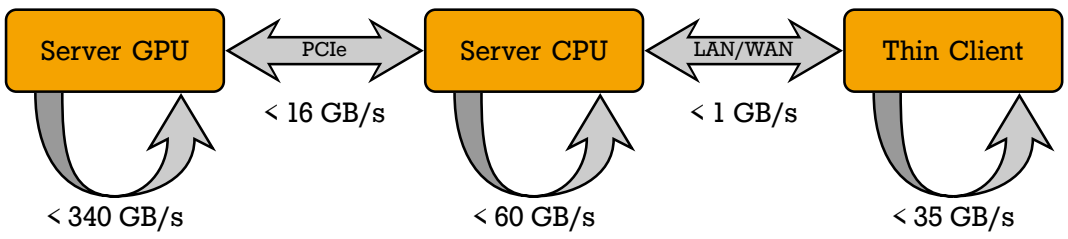


Figure 6.3.: Typical network, bus, and memory bandwidths relevant to streaming a GPU-based simulation. The two most limiting factors are the network bandwidth and the PCIe bus bandwidth.

A simulation post-processing service could provide a fixed set of derived values. However, the derived values a user wants to visualize often depend not only on the physics domain, but also on the application domain. Therefore, compiling such a fixed set requires domain knowledge and is very likely to be

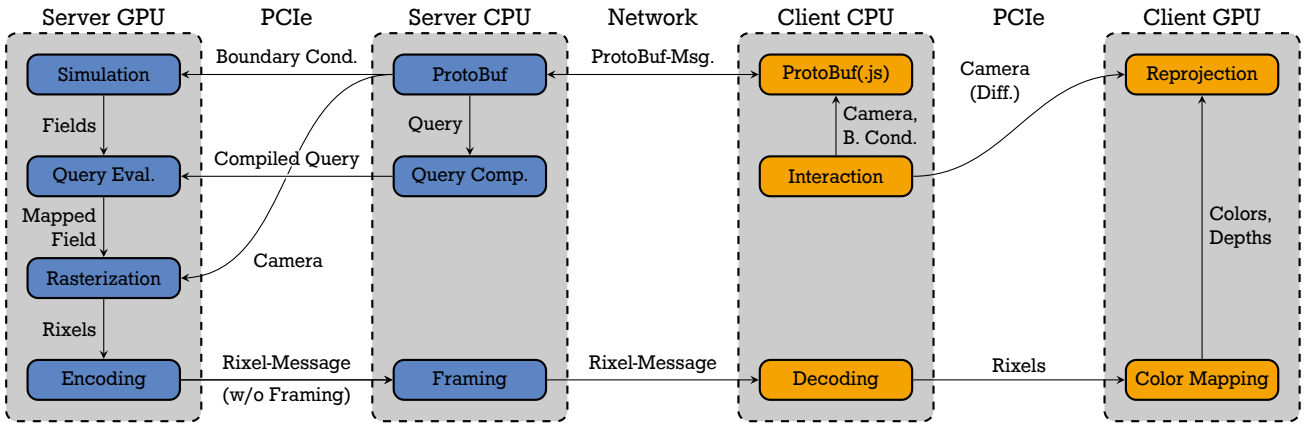


Figure 6.4.: Architecture block diagram showing how the query compiler can be integrated with rixel-based streaming. On mobile devices, client CPU and GPU are often on a single system on a chip (SoC) sharing a common memory bus, removing local PCIe overhead.

incomplete and insufficient for the user to perform his or her work. For stationary simulations, a server-side interpreter for user queries is entirely sufficient, as each query only has to be processed once. For interactive simulations, i.e., time-dependent simulations running at several frames per second, or the in-situ visualization of a long-running solver, however, this approach becomes costly due to the repeated interpretation overhead.

To avoid these costs, we examine the performance and bandwidth benefits of using optimizing compiler technologies for remote, in-situ post-processing and visualization of simulations running at interactive rates. In particular, to answer the fourth and final research question

4. **Can general purpose computing on the GPU (GPGPU) and code generation for the GPU be used to improve the performance of remote post-processing and visualization?** In particular, how can bandwidth overheads be minimized and GPU performance be exploited when user queries are only known at runtime?

we examine the following aspects:

1. Can the visualization of GPU-based simulations running at interactive rates profit from compiling queries to code running on the GPU?
2. Can compiler technologies be used to decrease visualization and interaction latencies in a remote scientific visualization system?

The implemented query compiler has a native CPU back-end (x86 and x86-64) as well as a GPU back-end (NVIDIA parallel thread execution (PTX)). The latter is used to extend the bandwidth savings to the PCIe bus in addition to the network interface, further improving visualization performance when using GPU-based simulation algorithms. The architecture block diagram in Fig. 6.4 illustrates which component runs on which device, and what data has to be transferred over which channel. Our approach is easily extended to all platforms supported by LLVM[2] [LA04b].

In this chapter, Section 6.2 discusses related work on the topic of compiler technologies for visualization, as well as compression and application sharing as supplemental techniques and alternative methods of

[2]The name “LLVM” is not an acronym, it is the full name of the project. See `https://llvm.org`.

improving or implementing streaming visualization. The concept and implementation of all components, especially the query compiler, the GPU-based simulation back-end, and the improved rixel encoding approach, are described in Section 6.3. The evaluation and our results are presented in Section 6.4. Finally, a summary of the chapter is given in Section 6.5.

6.2. Related Work

This section describes related work on code generation and compiler technologies for visualization, compression methods for scientific, floating point data with a focus on GPU-accelerated algorithms, as well as application sharing and video streaming. For a general background on sparse matrix data structures and GPGPU, refer to Chapter 2.

6.2.1. Compiler Technologies for Visualization

Previous applications of compilers and domain specific languages (DSLs) to scientific visualization mostly center on volume visualization and rendering itself [CKR+12; CCQ+14; RBGH14; KCS+16]. These systems therefore represent the entire visualization pipeline. In the streaming architecture presented in this chapter, data is transformed on the server and rendered on the client. Therefore, the aforementioned systems are not directly applicable.

This split corresponds to the two stages "Data Management" and "Picture Synthesis" in the system architecture used by Duke et al. [DBWR09]. However, they use an embedded DSL (eDSL) based on Haskell [Pey03]. As client code must be considered untrusted by the server, a general-purpose language and any eDSL based on such a language pose a great security risk. Furthermore, the complexity of existing methods which are aimed at efficiently implementing visualization algorithms makes them unsuitable for user-defined queries.

In the area of visual analytics, MapD Technologies [Map16] (since re-branded as OmniSci [Omn18]) have used LLVM/NVVM [NVI19] and GPU computing with great success to accelerate database queries [MŞ15; Mos18]. Since the publication of the paper on which this chapter is based, Ledur et al. have published a DSL for geospatial data visualization [LGMF17]. In contrast, we aim to bring the advantages of using compiler technologies to the field of scientific visualization, with a focus on interactively changing datasets from either in-situ or interactive simulations running on the GPU.

6.2.2. Compression

Another approach to reduce bandwidth requirements is to apply floating point data compression. For structured data, i.e., fields on regular n-dimensional grids as opposed to unstructured meshes, lossy methods such as the one presented by Lindstrom [Lin14] or the more recent method by Tao et al. [TDCC17] achieve good results. Structured data occurs in a significant subset of simulation domains and such a method would be widely applicable. However, lossy compression before calculation of desired derived values can lead to larger errors in the compounded result.

For general data, a method such as the one presented by O'Neill and Burtscher could be used [OB11]. They present a lossless compression algorithm for double-precision floating point data implemented on the

GPU, making it applicable to reducing network as well as PCIe bus bandwidths and to arbitrary simulation domains. As compression is orthogonal to the method presented in this chapter, any suitable compression algorithm can be chosen and combined with our approach. However, all compression methods incur an additional computation cost. Especially the cost of performing decompression in an browser-based client may be prohibitive (cf. deserialization cost in JavaScript in Section 6.4.2).

A good overview of existing compression techniques for floating point data is given by Ratanaworabhan [RKB06], showing compression ratios as well as compression and decompression times. More recent overviews are available in a preprint by Delaunay et al. [DCG18] and in a state of the art report on general data reduction techniques by Li et al. [LMG+18].

6.2.3. Application Sharing

Although we present a method to reduce the amount of data transferred when the client performs part of the necessary calculations to reduce perceived latency, it is worth mentioning that transmitting the content of single applications or the entire desktop as an image or video stream is still a common way to visualize server applications on (thin) client machines across a local network or the Internet. Microsoft's Remote Desktop Protocol (RDP) [Mic16] or the platform-independent Virtual Network Computing (VNC) [RSWH98] are two popular implementations of this concept.

Good results have also been achieved in the area of video streaming for games, chiefly through the use of hardware-based video codecs present on many modern GPUs [CCT+11]. Biedert et al. have applied hardware-accelerated video encoding in the area of distributed, tiled remote rendering using HPC clusters, achieving low latencies at high resolutions on local networks [BMFG18]. However, mobile networks, especially 3G networks, can add several hundreds of milliseconds of latency [Gri13]. Therefore, on lower performance networks such as mobile networks and the internet, approaches that decrease interaction latencies using hybrid rendering such as the method by Altenhofen et al. [ADSF16] or the method presented in this chapter become necessary.

As shown in this section, many approaches for remote visualization exist in the context of scientific visualization and visual analytics. However, the potential of compiler technology in the field of remote visualization of interactive simulations has not been discussed yet. Especially in modern HPC or cloud environments, these techniques can greatly improve usability by optimizing data transmission and increasing update rates on the clients, while minimizing server overhead and latency. Existing compression algorithms can be applied independently to decrease the required bandwidth even further. However, the resulting increase in encoding and decoding time has to be kept in mind and reduced to a minimum to achieve a net improvement in performance.

6.3. Concept and Implementation

In this section, we present our prototype visualization system, which consists of:

1. an interactive simulation back-end running on the server
2. a visualization front-end running on the client

3. an application-specific streaming protocol

4. the query expression compiler

Using the streaming protocol, simulation data is transmitted at interactive rates from the server to the client. By transmitting data instead of images, many interactions, for example color map changes, become possible on the client without incurring network round trip and transmission latencies. When the user wants to visualize values that are not a direct output of the simulation back-end, the query expression compiler is used to efficiently transform data on the server, reducing network bandwidth requirements. The prototype is based on a CFD simulation back-end, however, the method is directly applicable to other physical domains such as computational structural mechanics (CSM), which can be GPU-accelerated as described in Chapter 4, computational aeroacoustics or computational electrodynamics. For easy reuse with other simulation back-ends, the query compiler is designed as a shared library with a simple interface.

In the following, we briefly outline the visualization front-end and detail the simulation back-end, the streaming protocol, as well as the query compiler. Additionally, we describe our improvements to Altenhofen et al.'s rixel streaming approach [ADSF16].

6.3.1. Visualization Front-End

Two streaming clients have been implemented:

1. A graphical client running on a desktop machine shown in Fig. 6.5.

2. An HTML5+JavaScript client for streaming performance measurements shown in Fig. 6.1.

The former allows user interaction such as selecting the results to show, or entering an expression combining multiple result fields. Furthermore, the color mapping can be interactively modified by manipulating the color ramp widget with the mouse. The latter was developed to determine feasibility of a web client by evaluating streaming performance including deserialization. Both can be used to stream regular 2D and 3D grids. However, visualization is limited to 2D slices in the JIT-compiled streaming query prototype. Remote visualization with client-side navigation is described in Section 6.3.5.

6.3.2. Simulation Back-End

Our query-based streaming prototype is based on an interactive, GPU-accelerated, Eulerian 2D/3D-CFD code for staggered regular grids with cut cells using the multigrid solver we presented in an earlier publication [WMSF15]. All computational kernels are implemented in CUDA [NBGS08]. Therefore, GPU-CPU transfers are only required for data that is sent to the client.

To simulate incompressible fluids with constant density ρ it is necessary to compute a divergence-free velocity field $\mathbf{u}$. The incompressibility constraint can be enforced with a pressure projection step by determining a pressure field p which satisfies the Poisson equation

$$\nabla \cdot \nabla p = \frac{\rho}{\delta t} \nabla \cdot \mathbf{u}, \tag{6.1}$$

on a domain Ω subject to boundary conditions on $\partial\Omega$, where δt is the time step. We discretize the boundary

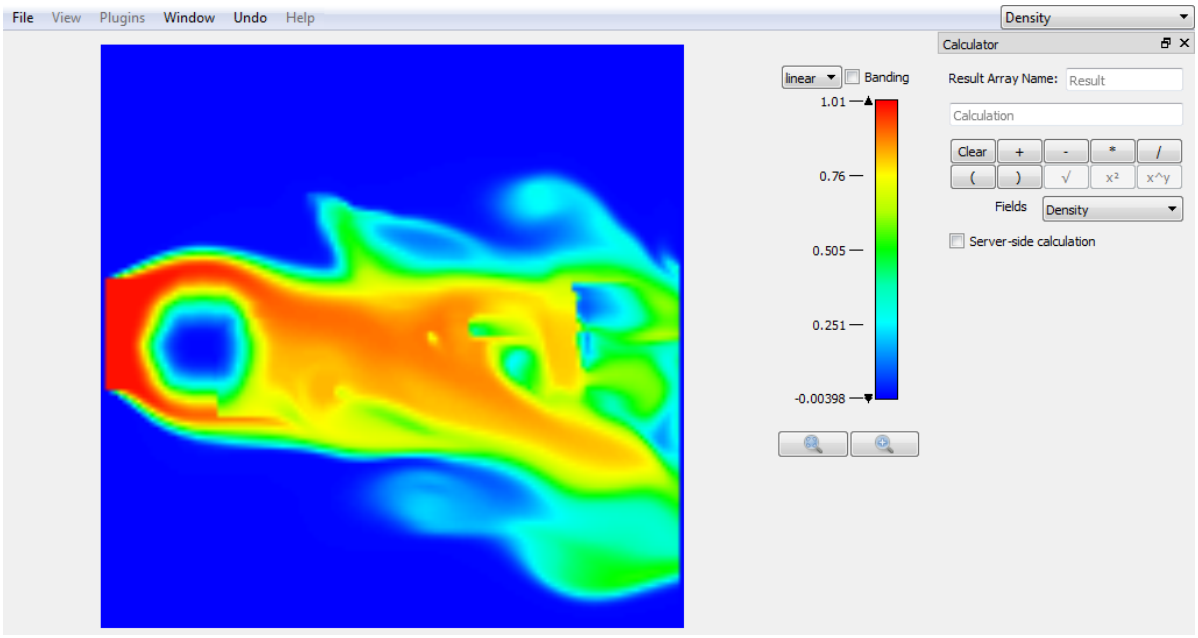


Figure 6.5.: The graphical streaming client. The user can choose which result field to view or enter an expression combining multiple fields. Color mapping can be modified interactively by clicking and dragging the color ramp widget.

value problem on a structured, orthogonal and equidistant staggered grid, as introduced by Harlow and Welch [HW65] and popularized in computer animation by Foster and Metaxas [FM96], where the pressure variables are located at the cell centers and the normal components of the velocities are located at the interfaces between adjacent cells. This setting is depicted in Fig. 6.6 and is described in detail in a textbook by Bridson [Bri08]. For visualization and streaming, the normal components of the velocity $\mathbf{u}$ are linearly interpolated to cell centers.

To avoid the stair-stepping issue in older grid-based methods (see, e.g., [Bri08]), we introduce a cut-cell formulation based on the FVM. We transform Eq. (6.1) into the weak form by integrating both sides and applying the divergence theorem. This leads to

$$\oiint_{\partial(C_{ijk}\cap\Omega)} \mathbf{n}\cdot\nabla p \,\mathrm{d}A = \frac{\rho}{\delta t}\oiint_{\partial(C_{ijk}\cap\Omega)} \mathbf{n}\cdot\mathbf{u}\,\mathrm{d}A, \tag{6.2}$$

where C_{ijk} is a voxel grid cell. Equation (6.2) is further discretized as

$$\sum_f A_f \frac{p_F - p_{ijk}}{\Delta x} = \frac{\rho}{\delta t}\sum_f A_f \pm u_f, \tag{6.3}$$

where f is a face of C_{ijk}. A_f and u_f are its face area and normal velocity, respectively, as indicated in Fig. 6.6. The sign of u_f depends on the orientation of the face. Finally, p_F is the pressure at the cell opposite to C_{ijk} along face f.

Compared to the cut-cell discretization by Ng et al. [NMG09], our discretization is more flexible due to the use of face areas instead of a signed distance function to approximate face areas. Furthermore, by employing volume-scaled restriction and prolongation operators in our multigrid solver, discretizing the system at lower resolutions results in the same system matrix as applying the Galerkin coarse grid method

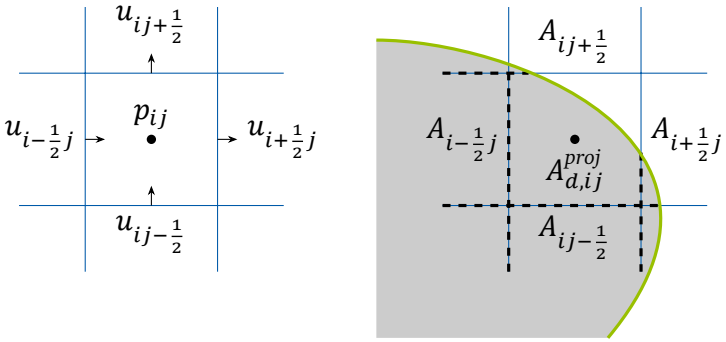


Figure 6.6.: Two-dimensional staggered grid showing the storage locations of pressures and velocities (left), as well as discretized areas (right). The arrows indicate the global orientation of the face normals. The dashed lines and gray background represent areas inside obstacles and the green line corresponds to the domain boundary $\partial\Omega$.

(see, e.g., [TS01]), leading to good convergence rates without requiring expensive general sparse matrix-matrix products (SpGEMMs) to compute the coarse systems. For more details, measurements, and a full derivation of our FVM discretization, please refer to the original publication [WMSF15].

6.3.3. Streaming Protocol

The streaming protocol is based on Protocol Buffers (ProtoBuf) [Goo08] for serialization and deserialization. ProtoBuf is a platform-independent open source framework that generates serialization and deserialization code from declarative message descriptions, which greatly simplifies modifications to the protocol. Implementations of ProtoBuf are available for a large number of programming languages, including C++ (as used by our server) and JavaScript (as used by the HyperText Markup Language 5 (HTML5) client). To minimize overhead, the fields of physical values are marked as packed repeated fields using `[packed=true]`. This prevents ProtoBuf from inserting type tags between each value and ensures that values are transmitted contiguously.

The generated messages are transmitted using the WebSocket protocol, as defined in RFC 6455 [MF11]. Although WebSockets are based on Transmission Control Protocol (TCP) and have a greater overhead than using User Datagram Protocol (UDP), they have several advantages. First, WebSockets ensure that message order is preserved and that all messages are received unless the connection is lost entirely, simplifying client and server implementation. Second, an increasing number of mobile applications are provided as an HTML5 web applications and WebSockets are supported by all current browsers, while TCP and UDP are not accessible from JavaScript. This ensures portability of our streaming solution to HTML5+JavaScript. A potential alternative would be to use WebRTC [HHE15], a newer standard designed for real-time communication. However, browser support remains at significantly less than 90% at the time of writing this thesis [DS19].

While streaming, the client sends frame request messages whenever a simulation time step (frame) is received, causing the server to send the most current time step that has been computed since sending the previous one. This ensures that no more messages are sent than can be transferred, which would lead to buffer overruns. To prevent bandwidth from being wasted due to the latency of requesting a new frame only after the previous one has been received, two frames are requested when a new connection is established, which corresponds to double buffering. A larger number of frames could be queued (triple

buffering or more) to overcome larger transient bandwidth changes. A full evaluation over varying buffer sizes was not performed within the scope of this work. Using the double buffering approach as described leads to an improvement in bandwidth exploitation of up to 50% compared to the naïve implementation.

Which fields are streamed to the client is determined by a query. The streaming prototype currently supports two query types:

1. Any number of result fields, e.g., `VelocityX` and `VelocityY`.
2. A query expression combining multiple fields into one.

The former is used when individual results are viewed by the user, and when post-processing is performed on the client for evaluation. The latter is forwarded to our query compiler or an interpreter that was implemented for comparison. A query expression consists of identifiers for the respective available results fields, operators or functions combining them, and parentheses for controlling operator order. The identifiers are specific to the simulation back-end and characteristic of the respective physical domain, e.g., `VelocityX`, `VelocityY` or `Pressure` for fluid simulations, or `DisplacementX`, `StressXX` or `StressXY` for structural mechanics simulations. These identifiers can be used to evaluate combinations of multiple fields such as `(VelocityX^2 + VelocityY^2) / 2 + Pressure`, which corresponds to $\frac{1}{2}|\mathbf{v}|^2 + p$, the sum of kinetic and static energy densities of a fluid with density $\rho = 1$.

6.3.4. Query Compiler

The query compiler prototype consists of an expression parser and an LLVM-based, optimizing back-end. Additionally, an interpreter has been implemented for comparison. The compiler is packaged as a shared library, for easy reuse on both client and server. While it would be simpler to generate CUDA or OpenCL code and either compile it via NVRTC [NVI18d] or OpenCL's runtime compiler, NVRTC does not support generating CPU code and OpenCL does not support interoperability with CUDA. Furthermore, in both cases, the expression would have to be rendered as text, including supporting loops etc. before being parsed again. By using LLVM directly, all code remains in memory as an abstract syntax tree.

For many optimizations, especially vectorization, the optimizer must have knowledge if pointers to data:

1. …may alias or not. Aliasing occurs if the same address in memory is reachable via different pointers. Aliasing prevents vectorization, as it can introduce additional dependencies between loop iterations if a pointer to data that is being read from can alias a pointer to data that is written to.
2. …are aligned or not. Aligned data is allocated with at an adress that is a multiple of a specific power of two. This information is relevant as many vector instruction sets require loads and stores to be aligned to achieve maximum throughput. On the GPU, this information can be used to issue wider loads when threads access more than one value (see Section 2.1).
3. …are captured or not. A captured pointer is stored somewhere and may later on be accessed via a different call. This is mostly relevant to callers of a specific function to know if a piece of data remains accessible.
4. …point to data that is read, written or both. This information is mostly relevant to callers who may want to reorder function calls.

Such information can be passed to LLVM via the use of function and parameter attributes. To maximize the number of optimization opportunities, the CPU back-end generates LLVM intermediate representation code (LLVM-IR) as an in-memory abstract syntax tree annotated with the appropriate parameter and function attributes according to the LLVM Performance Tips for Frontend Authors [LLV19] (see Listing 6.1). Specifically, annotating input pointers with the `readonly` and `nocapture` attributes and the output pointer with `noalias`. However, `nocapture` and `readonly` can be inferred by the compiler and did not affect optimization.

In previous LLVM versions, the use of `noalias` was necessary to ensure that vectorizing optimizations are not blocked by alias analysis. In current LLVM versions, vectorized code is generated independently of the presence of the `noalias` attribute. To do so, LLVM adds runtime aliasing checks and a non-vectorized version of the code. However, this increase in code size and the additional check showed no measurable effect on time measurements in our use case. At the time of writing the original paper on which this chapter is based, this feature was only available in the unstable development version of LLVM.

Additionally, alignment annotations (`align` n) can be used so that aligned moves are emitted instead of unaligned moves. Evaluations in a separate test environment with a result field of 4096^2 values did not result in any change in performance on either an Intel Xeon E5-2650 v2 CPU or an Intel Core i7-3770 CPU. In light of this result and as using alignment in the complete process would have required changes to the simulation algorithm's allocation strategy, alignment attributes were not used in the final evaluation.

For the GPU back-end, the LLVM NVPTX target was chosen. Alternatively, NVIDIA's proprietary NVVM-IR or OpenCL's SPIR could have been used, as both are based on LLVM-IR as well. NVVM-IR is used with `libnvvm` [NVI19], NVIDIA's compiler library. `libnvvm` supports additional proprietary optimizations, which can lead to improved performance. SPIR can be used with OpenCL to support both AMD and NVIDIA GPUs. However, both NVVM-IR and SPIR are based on older LLVM versions. Therefore, using either would mean using two different versions of LLVM for CPU and GPU code, or not having the full range of CPU optimizations, such as vectorization in the presence of potential aliasing, and instruction sets supported in current versions available. In the future, the addition of SPIR-V [KO18], the binary intermediate representation introduced with Vulkan and OpenCL 2.1, as an additional target for LLVM [Yax15] will make targeting all platforms that support OpenCL significantly simpler[3].

LLVM's optimization pipeline consists of a set of passes which take LLVM-IR as input and produce transformed LLVM-IR as output, as well as a number of analysis passes. One such pass is the instruction combining pass, which replaces complex instruction sequences by simpler instructions if possible. Among these are transformations that convert calls of math library functions such as `powf` to calls of faster functions such as `sqrtf` for `powf(`x`, 0.5)` or a single floating point multiplication for `powf(`x`, 2)`. However, these functions are identified by name and NVIDIA `libdevice` math library prefixes all names with `__nv`. To make full use of the instruction combining pass for GPU code as well, we generate code using unprefixed calls and run a subset of optimizations (primarily inlining and instruction combining) before retargeting call instructions to the prefixed versions and linking `libdevice`. After linking, the full set of optimization passes is run, which itself includes repeated passes of some analyses and transformations.

[3] As of September 2019, SPIR-V is only supported in Khronos' fork of LLVM `https://github.com/KhronosGroup/SPIRV-LLVM`

Listing 6.1: LLVM-IR generated by the query compiler before optimization for an expression equivalent to a saxpy-operation. The text representation of this code was only generated as a visualization, internally the LLVM-IR remains in the form of an in-memory abstract syntax tree.

```
; Function Attrs: alwaysinline nounwind readnone
define private float @kernel(float, float, float) #0 {
entry:
  %3 = fmul float %0, %1
  %4 = fadd float %3, %2
  ret float %4
}

; Function Attrs: nounwind
define void @map(i64, float* noalias nocapture, float, float* nocapture readonly,
    float* nocapture readonly) #1 {
entry:
  %5 = icmp ult i64 0, %0
  br i1 %5, label %body, label %exit

body:                                             ; preds = %body, %entry
  %6 = phi i64 [ 0, %entry ], [ %13, %body ]
  %7 = getelementptr inbounds float, float* %3, i64 %6
  %8 = load float, float* %7
  %9 = getelementptr inbounds float, float* %4, i64 %6
  %10 = load float, float* %9
  %11 = call float @kernel(float %2, float %8, float %10)
  %12 = getelementptr inbounds float, float* %1, i64 %6
  store float %11, float* %12
  %13 = add nuw i64 %6, 1
  %14 = icmp ult i64 %13, %0
  br i1 %14, label %body, label %exit

exit:                                             ; preds = %body, %entry
  ret void
}

attributes #0 = { alwaysinline nounwind readnone }
attributes #1 = { nounwind }
```

Unlike a general purpose, Turing complete programming language, the simple nature of our query expressions ensures that security is easy to maintain. A general purpose language would require sandboxing to disallow certain operations, and ensure that illegal code does not crash the entire system. Additionally, timeouts would be necessary to prevent infinite loops and/or deadlocks from affecting the server. Expressions with no explicit looping constructs and access only to mathematical functions are inherently secure. The only necessary limit is the length of the expression, as an arbitrarily long expression can result in an arbitrarily large amount of work.

6.3.5. Hybrid Rendering of 3D Data

Compared to streaming of regular 2D grids or slices of 3D grids to allow for client-side color mapping and navigation as described in the previous sections, remote visualization of 3D surfaces resulting from simulations running at interactive rates involves additional challenges. While navigation in a 2D grid, i.e., displaying a continuous rectangular subdomain, only involves zooming and panning, 3D navigation

Figure 6.7.: Streaming remote visualization of an interactive flooding simulation running at interactive rates. The simulation results are composited with the terrain geometry on the browser-based client described by Bormann et al. [BGM19].

involves varying camera positions, orientations, and projections. An example of an interactive 3D simulation visualized remotely is shown in Fig. 6.7.

By streaming geometry, unlimited client-side navigation becomes possible. However, when geometry continuously changes, as is the case in interactive simulations, the cost of transferring complete geometry can become prohibitive. While level of detail and progressive meshes ameliorate the situation when transferring static geometry, generating these typically involves significant pre-processing/encoding effort.

In Altenhofen et al.'s rixel approach [ADSF16], geometry transfer is handled by transmitting a colored point cloud of visible pixels. This allows for limited latency-free client-side navigation by reprojecting the previous point cloud, i.e.,

$$\mathbf{p}_n = \mathbf{P}_c \mathbf{V}_c \mathbf{p}_w, \tag{6.4}$$

where $\mathbf{p}_n$ is the resulting position in (homogeneous) normalized device coordinates, $\mathbf{p}_w$ is the world-space position of the point, $\mathbf{P}_c$ is the current projection matrix, and $\mathbf{V}_c$ is the current view matrix.

Client-side navigation by reprojection is limited by the fact that only geometry visible in the original view can be reprojected and that it results in varying resolution. However, initially obscured geometry becomes visible and full resolution is restored when a new frame is received, which happens several times per second.

Point visibility is handled on the server by rendering positions and colors using OpenGL [SA18]. The resulting buffers are read back to CPU memory and points with a depth less than the initial depth are sent to the remote client. However, transmitting 96 bits for the position (three 32-bit floating point coordinates) and 24 bits for the color of each visible point to the client incurs a high bandwidth cost. Even only transferring the same data for every viewport pixel from GPU to CPU can be costly (cf. Fig. 6.3).

If the original client-specified projection $\mathbf{P}_o$ and view $\mathbf{V}_o$ matrices are known, $\mathbf{p}_w$ can be determined from its original normalized device coordinates $\mathbf{p}_o$. This fact is exploited in depth image based rendering (see,

e.g., [MB16]) where $\mathbf{p}_o$ can be reconstructed using

$$\mathbf{p}_o = \begin{pmatrix} \frac{2x+1}{w} - 1 \\ \frac{2y+1}{h} - 1 \\ 2d - 1 \end{pmatrix}, \tag{6.5}$$

where x and y are the point's integer position in the image, w and h are the width and height of the image, and d is the stored depth value. This allows $\mathbf{p}_w$ to be reconstructed as

$$\mathbf{p}_w = \mathbf{V}_o^{-1}\mathbf{P}_o^{-1}\mathbf{p}_o \tag{6.6}$$

and then reprojected using Eq. (6.4). However, applying this approach directly would require transferring all viewport pixels, not only the visible ones, to the client. To avoid this cost, we compute a one bit per viewport pixel mask (a bitmap) and only transfer depth values (16-bit or 32-bit, depending on precision requirements) for valid rixels, e.g., foreground/rendered rixels. The client can then use the mask to reconstruct each rixel's position in the image. Additionally, we transmit the physical values of the currently selected field instead of a color. This allows for client-side changes of color mapping settings without latency. By transferring this data as a 16-bit fixed point value per rixel along with a 32-bit floating point minimum and maximum per frame, bandwidth costs are reduced further. Besides reducing message size, rendering fillrate requirements are reduced, as depth testing is always required but the additional render target for positions can be omitted.

For the simulation server, a GPU-based shallow water solver was implemented based on the work of Brodtkorb et al. [BSA12] and Vacondio et al. [VPM14], as described in Lotter's BSc thesis [Lot18]. We use a second order in time and space discretization on a regular grid, using an explicit, adaptive time step integration method. While the state vector and all intermediates are allocated at the full size of the domain, only a small percentage of the domain contains water at any point in time. As an optimization, only 16×16 subdomains that contain water, or are adjacent to cells that contain water, are considered in the simulation kernels. Combined with the fast, GPU-parallelized solver, this allows us to achieve simulation rates well beyond real time suitable for prediction.

The shallow water simulation is implemented in CUDA [NVI18a], which offers OpenGL interoperability functions. Therefore, the results of the simulation, potentially mapped using a user query, can be used for initial server-side rasterization without copying data between GPU and CPU (cf. Fig. 6.4). We additionally perform mask computation, stream compaction (removal of invalid viewport pixels), and minimum/maximum reductions on the GPU using CUDA and Thrust [BH15] to additionally reduce PCIe bandwidth costs to a minimum. The CPU only adds a message identifier and WebSocket framing. The message identifier is used by the client to associate the correct viewport size and matrices with the received frame.

6.4. Results

In this section, we analyze the performance of our streaming protocol and our query compiler. For the evaluation, the simulation server was set up on a dual Intel Xeon E5-2650v2 server (two octa-core processors running at 2.66 GHz) running Ubuntu Linux 13.10 with 64 GiB DDR3-1866 and two NVIDIA GRID K2 graphics cards with 2 GPUs and 8 GiB GDDR5 each. The graphical client was installed on an Intel Core

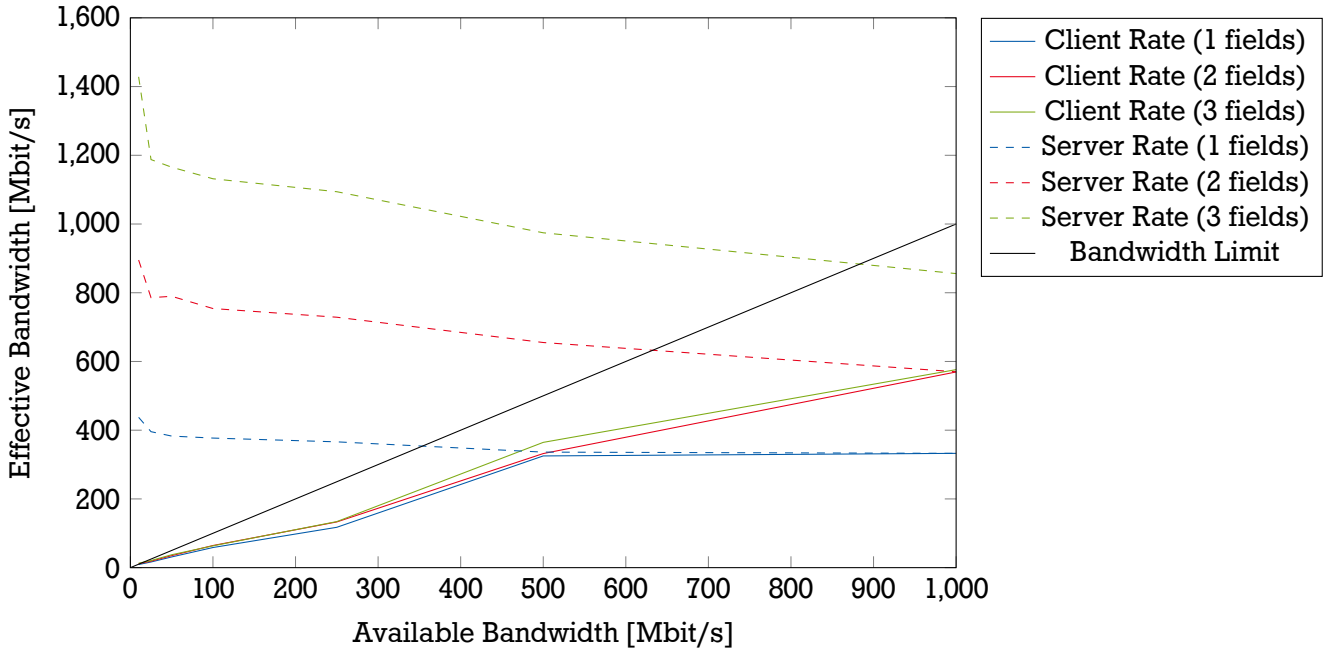


Figure 6.8.: Effective client bandwidths when network bandwidth is limited. The rate at which the server produces data imposes an additional upper limit. This limit decreases with increasing network bandwidth as the server spends more time serializing data and is only reached for bandwidths greater than 500 Mbit/s per field.

i7-2600 (quad-core processor running at 3.4 GHz) desktop workstation running Windows 7 with 16 GiB DDR3-1333 and an NVIDIA GeForce GTX 580 GPU with 1.5 GiB GDDR5.

For the HTML5 client, tests were additionally performed on a OnePlus One smartphone with a Qualcomm Snapdragon 801 CPU (quad-core processor running at up to 2.5 GHz) and 3 GiB LPDDR3-1866 running Cyanogen OS 12.1 (based on Android 5.11). To cover both major mobile platforms, tests were also performed on an Apple iPhone 6S with an Apple A9 CPU (dual-core processor running at up to 1.85 GHz) and 2 GiB LPDDR4-1600 running iOS 9.2.

In the following, Sections 6.4.1 to 6.4.3 are evaluated on the basis of the 2D client (see Fig. 6.1) and therefore all fields are transmitted as dense grids, without any rasterization or reprojection using OpenGL. The improved rixel streaming approach introduced in Section 6.3.5 is evaluated separately with respect to bandwidth cost in Section 6.4.4.

6.4.1. Network Performance and Bandwidth Limitations

Figures 6.8 and 6.9 show the system's performance in terms of data throughput and frames per second when transmitting one, two or three fields with different network bandwidths. In this particular example, these fields were `Pressure`, `VelocityX` and `VelocityY` with a size of 1024^2 floating point values each. Bandwidth limiting was realized on the server side using Linux Traffic Control `tc`. Only outgoing bandwidth is limited, but the messages sent by the client are only tens of bytes in size and should therefore not affect the results significantly.

Increasing the available network bandwidth also increases the client's data throughput as well as the achievable frames per seconds, as more data can be transmitted across the network. At the same time,

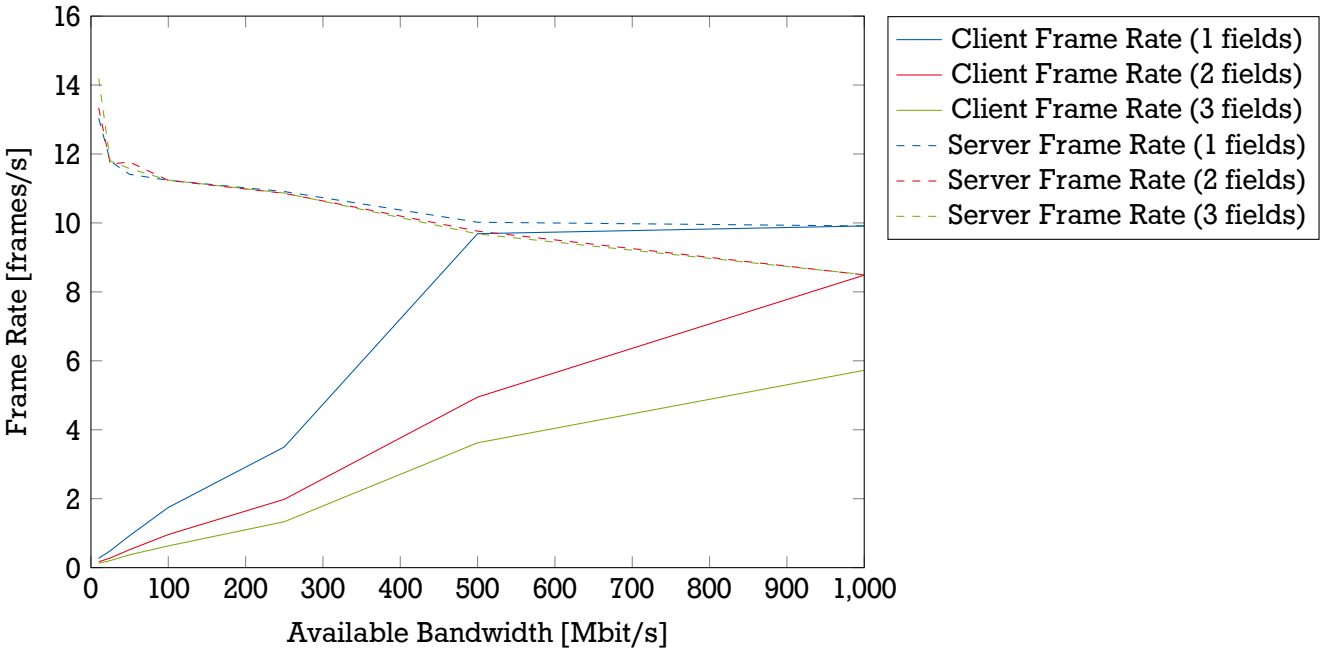


Figure 6.9.: Client and server frame rates for several network configurations. The client frame rate increases with higher network bandwidth, as more data can be sent by the server. As in Fig. 6.8, the server frame rate limits the client frame rate.

the server's throughput and frame rate drop slightly, because more time is spent serializing messages instead of calculating new results. This decrease could be partially compensated by implementing double buffering and overlapping simulation and serialization. However, this would lead to increased memory requirements.

In all cases, the server's performance is a natural upper limit for the client that cannot be exceeded. When transmitting more than one field, this limit only becomes relevant for client-server configurations in a LAN setup with more than 1 Gbit/s. For a single field, 500 Mbit/s are sufficient to reach full performance (cf. Figs. 6.8 and 6.9). The fixed bandwidth limit itself is never reached, as the limit is applied at the TCP level and the effective bandwidth only includes floating point data and neither control messages nor WebSocket and ProtoBuf encoding overheads. A direct comparison with VNC is difficult, as various VNC servers offer differing compression algorithms to transfer screen images, from uncompressed 32-bit colors, over lossless compression codecs, to lossy compression codecs with potentially very low quality. While the worst case (uncompressed 32-bit colors) corresponds directly in bandwidth requirements to a single field of 32-bit single precision floating point values (assuming field resolution corresponds to screen resolution), all interaction incurs a full round trip of latency, including image compression and decompression.

6.4.2. Serialization and Deserialization Costs

Another criterion for good performance and smooth visualization is the time required to serialize the results produced on the server and to deserialize the incoming messages on the client. Table 6.1 shows the serialization and deserialization costs for one, two, and three dense fields with a size of 1024^2 floating point values per field (as in Section 6.4.1). Each measurement represents an average over 500 simulation steps. Note that new frames are only transmitted to the client if the processing of the previous frame is finished. For the client, we tested both desktop and mobile environments with several browsers and the

Table 6.1.: Serialization and deserialization times for various platforms for a varying number of fields. Even on desktop machines, deserializing a single dense 1024^2 field in JavaScript takes approximately 0.1 seconds (86.5 ms on Chrome, 115 ms on Firefox).

	Time [ms]		
Number of Fields	**1**	**2**	**3**
Serialization	8.81	18.9	30.0
Native C++ (Desktop)	7.89	14.5	20.2
Chrome 47.0 (Desktop)	86.5	174	254
Firefox 42.0 (Desktop)	115	221	380
Chrome 46.0 (Android)	435	841	1202
Safari 601.1 (iOS)	233	346	516

native client. As all fields are concatenated for serialization, the required time increases linearly in all cases. While serialization and deserialization take between 7 and 30 milliseconds when using ProtoBuf in a native C++ application, performance decreases significantly when switching to browser-based applications using JavaScript. Although Chrome 47.0 outperforms Firefox 42.0, deserialization times of 86 to 254 milliseconds on a desktop workstation make it challenging to reach interactive frame rates ($\geq$ 10 frames per second) for more than one field.

On mobile devices, deserialization times of 435 and 233 milliseconds for Chrome 46.0 and Safari 601.1, respectively, make interactive frame rates effectively impossible, even on fast networks, and raise the need to investigate alternative (de)serialization methods (see Section 7.1.3).

6.4.3. Query Compiler

To analyze the performance of our query compiler, we measured average compile and evaluation times for three query expressions varying in complexity and number of result fields involved:

1. The absolute pressure $|p|$:
   `abs(Pressure)`
2. The absolute velocity $|\mathbf{v}|$:
   `sqrt(VelocityX^2+VelocityY^2)`
3. The total energy density $\frac{1}{2}|\mathbf{v}|^2 + gz + \frac{p}{\rho}$ with $g = 0$ and $\rho = 1$:
   `(VelocityX^2+VelocityY^2)/2 + Pressure`

These expressions were compiled and executed on the server described at the beginning of this section. Although this set of example expressions is not exhaustive, it consists of common expressions entered by a user while evaluating fluid simulations. The absolute value of the pressure can be of interest when the results of a compressible simulation are viewed, as the amplitude of an approximately periodic wave may be of greater interest than its absolute phase. As the magnitude of a vector field, the absolute velocity is frequently required and most visualization systems include it as a built-in option. The isocontours of the total energy density are an alternative to streamlines, as according to the Bernoulli equation the total energy density must remain constant along each streamline for incompressible fluids.

All measurements in this section were performed and averaged over 80 simulation runs on a 1024^2 grid running for 500 frames for each expression. Note that calculation is only performed for frames actually

Table 6.2.: Average compile and execution times for the three example expressions in Section 6.4.3. The times for the interpreter only include expression parsing.

	Compile Time [ms]			Execution Time [ms]		
Expression	**Interp.**	**CPU**	**GPU**	**Interp.**	**CPU**	**GPU**
Expr. 1	0.03	6.60	77.1	14.1	8.91	4.49
Expr. 2	0.04	7.22	77.1	53.1	13.1	4.27
Expr. 3	0.04	9.37	77.2	63.1	17.1	4.38

Table 6.3.: Decomposition of execution time into calculation and GPU-CPU transfer times. These measurements were performed on a different system than the one used for Table 6.2.

		CPU		GPU	
Expression		**Calc.**	**Copy**	**Calc.**	**Copy**
Expr. 1	ms	1.99	1.04	0.10	0.98
	%	65.7	34.3	9.2	90.8
Expr. 2	ms	2.20	1.98	0.13	0.97
	%	52.6	47.4	11.6	88.4
Expr. 3	ms	1.13	3.02	0.16	0.99
	%	27.2	72.8	13.6	86.4

transmitted to the client and that compilation is performed once per simulation run. Therefore, the sample size for the average compilation time is 80 per expression and less than 40000 for the average calculation time.

The measured compile and execution times are shown in Table 6.2. Comparing the execution time of the queries compiled for the GPU with using an interpreter, a speedup of more than 14× is achieved. CPU compilation is completed within less than 10ms and only shows a slight increase depending on expression complexity. Although marginally slower compilation is expected due to the repetition of some optimization passes (see Section 6.3.4), GPU compilation is much slower at over 77 ms and is dominated by a constant component. Further analysis shows that 33% of that time is spent linking `libdevice` and 61% is spent on the final set of optimization passes. A likely reason for the significant increase in optimization time is the much larger module due to the size of `libdevice`.

It can be seen from the execution times in Table 6.2 that the timings for the GPU version are approximately constant for all three expression, whereas for the CPU version they grow with the number of fields used in the expression. To determine the reasons for this behavior, additional measurements decomposing the total evaluation time into computation and data transfer times were performed. Table 6.3 shows the results of these measurements that were performed on a desktop machine equipped with an Intel Core i7-3770 CPU with 3.40 GHz and an NVIDIA GeForce GTX 580 GPU. In the case of CPU evaluation, all relevant fields have to be copied from the GPU depending on the expression used. This is reflected in the linear increase in copy times and explains the dependency seen in Table 6.2. For GPU evaluation, only the derived field has to be copied to system RAM. As the GPU evaluation times are dominated by the expression-independent copy component, the total execution time is approximately constant, as seen in Table 6.2, and leads to an improvement of up to 72.3% for Expr. 3. The calculation time without copying is up to 20× faster when using queries compiled for the GPU than for the CPU, as shown in Table 6.3 for Expr. 1.

Table 6.4.: Break-even points of using CPU or GPU JIT compilation instead of an interpreter and GPU instead of CPU JIT compilation for the three example expressions in Section 6.4.3. The numbers are computed from the measurements in Table 6.2 and rounded up to the nearest integer. Break-even before rounding is shown in parentheses.

Expression		Break-even [Frames] CPU	GPU
Expr. 1	Interp.	2 (1.27)	9 (8.02)
	CPU	—	16 (15.95)
Expr. 2	Interp.	1 (0.18)	2 (1.58)
	CPU	—	8 (7.91)
Expr. 3	Interp.	1 (0.20)	2 (1.31)
	CPU	—	6 (5.33)

The total time to process n simulation frames (time steps) is $t_c + nt_e$, where t_c is the compilation time, t_e is the average execution time per frame and n is the number of frames executed. Therefore the break-even between two methods a and b can be computed as $n = \left\lceil \frac{t_{c,a} - t_{c,b}}{t_{e,b} - te,a} \right\rceil$. Table 6.4 summarizes the different break-even points of using just-in-time (JIT) compilation instead of an interpreter. In all but the first case, the cost of compilation for CPU is amortized within the first frame, as the sum of compilation and execution time for one frame is less than the execution time for the interpreter. Due to the large compilation overhead, the break-even point of using the GPU instead of the CPU occurs significantly later. The break-even point of using the GPU instead of the CPU is reached after less than 10 frames for Expressions 2 and 3. As compilation time is independent of field size and execution time depends linearly on it, the break-even point will be reached even more quickly for larger simulation domains.

6.4.4. Hybrid Rendering of 3D Data

We compare the frame sizes resulting from our improved rixel encoding presented in Section 6.3.5 with the original encoding by Altenhofen et al. [ADSF16] in Fig. 6.10. The comparison is performed for a viewport size of 1024 × 1024 and using 32-bit depth values. For frames with a coverage of 10% or more, network message size is drastically reduced by up to 59% (73% if 16-bit depths are sufficient, not shown). For frames with very low coverage the bitmap can be larger than the original encoding. However, the break-even in Fig. 6.10 occurs at a coverage of less than 1.4% (less than 1.2% when 16-bit depths are used). Furthermore, the absolute size of the bitmap is only 128 KiB for a 1024 × 1024 frame.

In the original approach by Altenhofen et al., PCIe bus transfer size is constant, as the complete frame buffers are transferred from GPU to CPU, where invalid points are removed [ADSF16]. By contrast, we perform encoding and stream compaction on the GPU. Therefore, the frame size for both the network message and the PCIe bus transfer are identical. Compared to the original approach, transfer sizes are reduced by at least 59% (73% for 16-bit depths, not shown).

6.5. Summary

In summary, we have shown that compiler technologies for GPUs and CPUs can be used to significantly decrease visualization latency in remote scientific visualization. By using the query compiler introduced in Section 6.3.4, only one result field has to be sent to the client. This ensures that a high visualization frame rate can be achieved with bandwidths as low as 500 Mbit/s (see Section 6.4.1), allowing the user

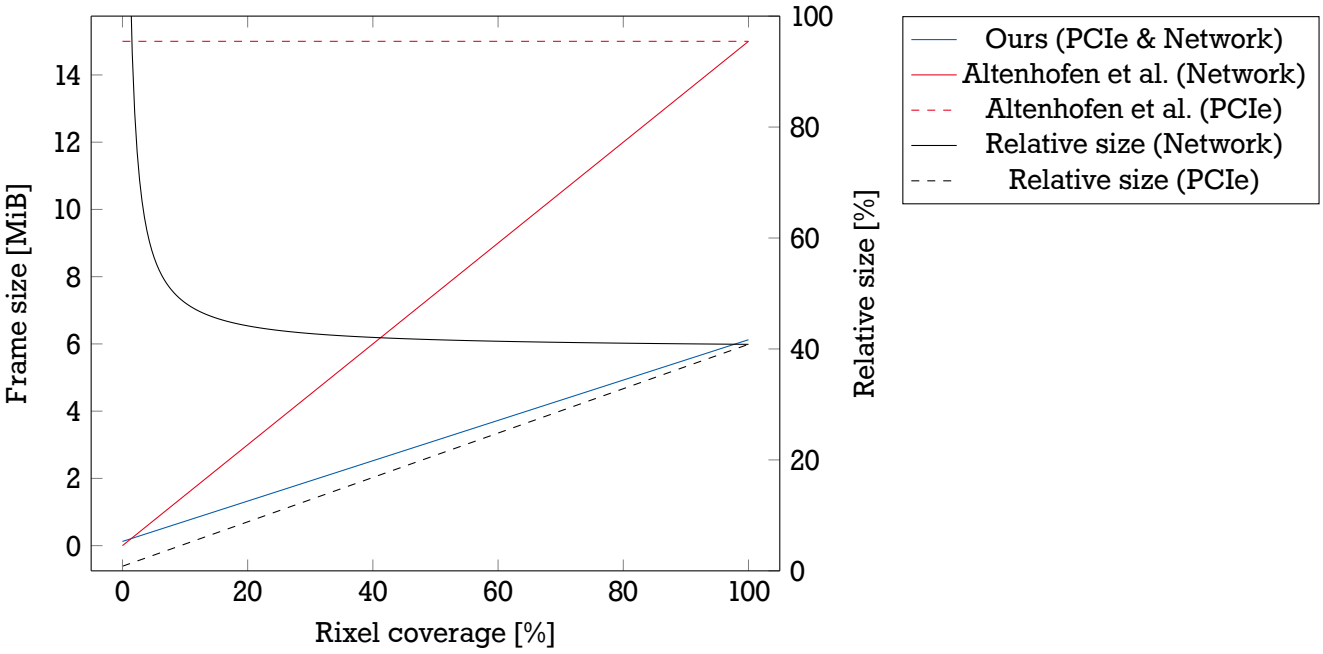


Figure 6.10.: Comparison of frame sizes over varying coverage, i.e., percentage of valid points, resulting from our rixel encoding and Altenhofen et al.'s [ADSF16] original encoding for a 1024 × 1024 viewport. Both network message size and transfer size over the PCIe bus are compared.

to view more current data. Additional latency and computation costs due to deserialization are avoided as well, making HTML5 clients feasible on desktop workstations (see Section 6.4.2). By using an optimizing compiler, server CPU and GPU times are reduced, leading to a speedup of up to 14× compared to the naïve approach of using an interpreter (see Section 6.4.3). Furthermore, we have shown that the visualization of GPU-based simulations running at interactive rates can significantly profit from query compilation to GPU code. By computing derived expressions directly on the GPU, a significant amount of time can be saved. By only copying a single field independent of the number of fields used in the expression, the amount of data transferred over the PCIe bus can be reduced, as shown in Section 6.4.3. Additionally, calculation speed is increased by a factor of up to 20× compared to the CPU (see Table 6.3).

We have performed a detailed analysis of streaming performance and shown that optimizing compiler technologies such as LLVM can be used to significantly improve performance and reduce bandwidth costs for streaming visualization of interactive simulations. By additionally moving data transformation work to the GPU, the costs of PCIe bus transfers can be minimized as well for GPU-based simulation back-ends. By introducing a new depth-based encoding of rixel data and performing stream compaction on the GPU (see Section 6.3.5), we similarly reduce network and PCIe bus transfer costs when visualizing 3D simulations running at interactive rates.

With respect to the research questions posed in Chapter 1, this chapter answers the fourth and final sub-question:

4. **Can GPGPU and code generation for the GPU be used to improve the performance of remote post-processing and visualization?** In particular, how can bandwidth overheads be minimized and GPU performance be exploited when user queries are only known at runtime?

We have shown that code generation and optimizing compiler frameworks can be used to more efficiently implement user queries in the visualization/analysis stage. In particular, by combining information on the GPU before transfer to server CPU memory, PCIe bus transfer costs are minimized, leading to reduced visualization latencies. Compared to performing combination on the client, latencies are reduced even more as the corresponding network transfer costs are avoided.

Compared to MapD (see Section 6.2.1) we have taken a similar approach of leveraging compiler technologies for visualization, but applied it to interactive or in-situ scientific visualization instead of visual analytics. The range of available options is currently smaller, but further enhancements are outlined in Section 7.1.3.

Compared to application sharing (see Section 6.2.3) our approach of pre-transforming simulation data on the server before transmitting it to the client for final visualization has both benefits and drawbacks. Many interactions relevant during exploration of simulation results, including color map changes and panning/zooming in 2D or limited (as described Section 6.3.5) changes in camera position in 3D, can now be performed without any network round trip latency using our approach. Application sharing always incurs at least one network round trip for all user interactions. However, the time to first image is potentially higher, as floating point simulation data is frequently larger than the resulting image compressed using a video codec. This also decreases the number of frames per second that can be transmitted given a limited bandwidth. This drawback can be offset by applying compression methods as well (see Section 6.2.2). Furthermore, the portion of simulation data that is transmitted could be limited to the visible part and resolution, however this limits panning/zooming or can create temporary holes in the visualization that are fixed as soon as an updated frame is received.

7. Conclusion

Throughout this dissertation, we have examined a selection of approaches to accelerating the core components of the virtual prototyping cycle by making better use of manycore graphics processing units (GPUs). The novel methods and algorithms presented in this thesis provide significant speedups in the areas of modeling, simulation, and visualization.

At the same time, the newly developed GPU-optimized mesh and sparse matrix data structures significantly reduce memory requirements compared to the state of the art, while the improved remote visualization methods reduce required bus and network bandwidths. Therefore, the presented techniques are not only more efficient in time, reducing power costs, but also in space, enabling the use of larger models on commodity hardware.

A complete list of contributions, including speedups and improvements in memory and bandwidth use, is given in Section 1.1 and discussed in more detail in the individual chapter summaries in Sections 3.6, 4.6, 5.5, and 6.5. All contributions have previously been featured in the peer-reviewed publications listed in Section 1.1.5.

Besides the use in virtual prototyping and other engineering workflows, the presented improvements beyond the state of the art are useful in other application areas as well. For example, multi-resolution modeling and physically based animation, popular methods in computer graphics, can benefit from the advances in parallel mesh processing, efficient cut cell fluid simulation, and faster finite element method (FEM) system matrix assembly. Furthermore, many of the improvements carry over to multicore system processor (CPU) parallelization as well, such as in the case of the volumetric mesh data structure introduced in Chapter 3 or the query compiler presented in Chapter 6.

Returning to the research questions given in Chapter 1, we first look at the four sub-questions of our main research question. The first two correspond to improving general purpose computing on the GPU (GPGPU) data structures and algorithms as an approach to making better use of the GPU. The latter two correspond to the code generation approach.

1. **Can the GPU be used to efficiently process unstructured meshes, both polyhedral meshes in general and tetrahedral meshes in particular?** If yes, which mesh data structures and algorithms are suitable for GPU processing?

    - By combining the concept of boundary and coboundary operators originating in algebraic topology (see, e.g., [Hat02]) with the novel compact ternary matrix storage format ternary compressed sparse row (TCSR) presented in Section 3.4, a very compact (up to 31% smaller than the state of the art) description of arbitrary, potentially non-manifold, polyhedral meshes is achieved that allows for efficient parallel access to mesh connectivity [MAS17].

    - Due to the matrix-based representation, many of the concepts of GraphBLAS, the description of graph algorithms in terms of linear algebra (see, e.g., [MBB+13]), can be applied to mesh processing as well. Sections 3.4.1 to 3.4.3 show how neighborhood queries, boundary extrac-

tion, mesh smoothing, and subdivision can be efficiently implemented, achieving speedups of up to more than two orders of magnitude ($> 531\times$) [MAS17].

- By encoding element orientation in vertex order and using implicit row offsets, even more compact representations are possible for tetrahedral meshes and other homogeneous simplicial complexes, e.g., purely triangular or tetrahedral meshes, while retaining the benefits of the novel mesh data structure, as shown in Section 3.4.4 [MS18].

2. **Can these GPU-optimized data structures be used to perform system matrix assembly for the FEM and other simulation methods more efficiently?** If yes, how can memory overhead be reduced while maintaining or improving performance?

   - By limiting the input to triangular or tetrahedral meshes (or other lower- or higher-dimensional simplicial complexes), which are favorable due to their more robust generation, FEM system matrices for meshes of arbitrary polynomial order can be allocated exactly with minimal topological information as shown in Section 4.3. In combination with the simplex mesh data structure in Section 3.4.4, FEM system matrix assembly including the determination of the sparsity pattern can be performed significantly faster (approximately $4.4\times$) than with current state of the art methods while using a significantly smaller amount of memory (as little as 14.3%), enabling the simulation of larger models [MS18].

3. **Can code generation and compiler techniques be used to efficiently implement GPU sparse matrix formats and algorithms required in simulation and mesh processing?** Specifically, how can the performance of sparse matrices with extended number systems (e.g., complex numbers and quaternions) and dense blocks be improved?

   - By jointly optimizing parallel schedules and sparse matrix layouts, sparse matrix operations on matrices with compound entries, e.g., extended number systems or dense blocks, can be performed more efficiently (speedups up to $4.7\times$) than with the highly tuned vendor library, as shown in Chapter 5. This is achieved by applying an autotuning approach using a code generator to compose schedule and layout variants. Matrices with complex or quaternionic entries occur in geometry processing, while dense blocks occur in the FEM [MSF19].

4. **Can GPGPU and code generation for the GPU be used to improve the performance of remote post-processing and visualization?** In particular, how can bandwidth overheads be minimized and GPU performance be exploited when user queries are only known at runtime?

   - By using an optimizing query compiler, latency in remote visualization can be significantly reduced, as described in Section 6.3. This results both in reduced computation times compared to an interpreter and more significantly in the GPGPU context reduced bus bandwidths (the amount depends on the expression). For 3D visualizations, our improved depth-based rich pixel (rixel) encoding and GPU implementation further reduce bandwidth requirements (by up to 73%). Reducing field processing times is especially valuable when visualizing interactive or in-situ simulations [MA16].

In summary, we have shown how both simplicial complexes and arbitrary polyhedral meshes can be efficiently represented and processed in parallel on the GPU using data structures based on sparse matrices and how FEM system matrix assembly can be performed efficiently by using them. Furthermore, we have

demonstrated how code generation and compiler techniques can be used to accelerate the virtual prototyping process on the GPU by applying input-aware optimizations to matrix data structures and by making the evaluation of user queries more efficient.

In conclusion, we can derive an answer to the main research question of this thesis:

> **Can the available hardware, particularly manycore GPUs, be used more efficiently in the virtual prototyping cycle?** If yes, which components of the virtual prototyping cycle can be improved in what manner?
>
> - By using the complementary approaches of improved GPGPU data structures and algorithms as well as code generation, many components of the virtual prototyping cycle can be performed more efficiently on the GPU than previously possible. Specifically, mesh processing during modeling, system matrix assembly in simulation, sparse matrix computations in modeling and simulation, and processing of user queries in remote visualization can significantly benefit from the presented methods. Furthermore, due to the similarities between manycore GPU single instruction multiple thread (SIMT) and modern multicore CPU single instruction multiple data (SIMD) architectures (see Section 2.1.1), many of the techniques can be applied to more efficiently using the CPU as well.

Having answered all research questions we posed at the beginning of this dissertation, let us take a step back to reflect on the long-term value of the contributions presented in this thesis. Will the approaches continue to be beneficial on future GPU architectures? Will future CPUs overtake GPUs in performance per watt again, as they recently have in deep learning inference speed[1] [STD+19]? While we cannot predict how future CPU and GPU architectures will evolve with certainty, we can make some conjectures based on recent developments.

For example, the current high end server processor Intel Xeon Platinum 9282, the processor used by Shen et al. [STD+19], has 56 cores, equaling the number of streaming multiprocessors (SMs) on the NVIDIA Quadro GP100 used in the evaluations in Chapters 4 and 5. Both use simultaneous multithreading (SMT) to hide memory latency, and while one uses SIMT with 32-thread warps, the other uses 512-bit wide (16 single precision floating point numbers) SIMD instructions with masking, which can be efficiently programmed using the same programming model (see Chapter 2). Therefore, this apparent[2] convergence in CPU and GPU architectures means that many techniques previously necessary only on GPUs will become necessary on CPUs as well for peak performance. Consequently, the data structures and algorithms presented in this dissertation are expected to become even more relevant beyond the scope of GPGPU only, as evidenced by the large CPU speedups in Chapters 3 and 6.

The data structure tuning and just-in-time (JIT) code generation approaches in Chapters 5 and 6 are future proof by design, as they tune or compile code for the hardware currently in use. It is merely necessary to update the underlying compilers to support the new hardware. However, on the one hand, the benefits of reducing peripheral component interface express (PCIe) bandwidth costs in Chapter 6 vanish when no external co-processor such as a GPU is used. With the increasing integration of CPUs and other co-

[1] However, the thermal design power is 400W instead of 300W for a 0.4% increase in speed.
[2] Some significant differences such as branch prediction on CPUs and specialized rasterization and recently ray tracing hardware on GPUs are expected to remain.

processors on systems on a chip (SoCs), these benefits may well decrease or vanish even when using GPUs in the near future. On the other hand, the network bandwidth and latency reductions will continue to be beneficial, as network bandwidth and latency are limited by physical distance, corresponding signal attenuation, and the speed of light.

7.1. Future Work

While the presented techniques offer significant speedups and memory savings compared to the state of the art in all major components of the virtual prototyping cycle, ample opportunity for further research remains. In addition to the potential research directions listed in the following subsections, reducing iteration times by speeding up calculations is not the only way to reduce prototype or product development time. Especially in light of the widespread availability of additive manufacturing (AM), improving user interfaces in computer-aided engineering (CAE) tools to make them easier to use and understand offers many opportunities for user experience and visualization researchers.

The following subsections are based on and expand upon the future work sections of the publications listed in Section 1.1.5.

7.1.1. Modeling

While the results in the area of mesh processing for modeling shown in Chapter 3 are extremely promising, several potential avenues for future research remain. Using TCSR to represent sparse matrices is very compact, but GPU memory coalescing can affect performance significantly, as explained in Section 2.1. Due to the irregularity of the boundary operator matrices and even more so the coboundary operator matrices, a ternary version of the Bin-CSR format for highly irregular matrices on GPUs presented by Weber et al. [WBS+13] may provide improved performance. However, the need for padding (see Sections 4.2.1 and 4.4.2) means either that zero-values must be explicitly representable, or that row lengths must be stored explicitly. In both cases, the encoding would become significantly less compact.

Alternatively, more efficient ways to operate on compressed sparse row (CSR) matrices such as LightSpMV by Liu and Schmidt [LS15] and the efficient general sparse matrix-matrix product (SpGEMM) for irregular matrices by Liu and Vinter [LV14] could be adapted for processing of TCSR matrices. In combination with replaceable field operators as used in GraphBLAS [MBB+13], operations currently requiring custom kernels could be implemented efficiently using a linear algebraic approach, improving extensibility and simplifying implementation.

Besides possibilities to further improve performance, another avenue of research are in-place topological changes, such as adding a layer of cells at the outer surface. These would benefit from a matrix format designed for dynamic updates on the GPU, such as the DCSR format presented by King et al. [KGKM16].

Finally, the presented data structures and mesh processing primitives form the basis for parallel volumetric modeling on the GPU. To gain full advantage of these primitives, they must be combined with additional operations into interactive modeling workflows. Their use to implement efficient parallel adaptive tetrahedral mesh booleans has recently been examined in the context of a master's thesis under my

supervision [Str19]. In that context, adaptive mesh refinement and intersection have been shown to work well on the GPU using the data structures presented in this dissertation, supplemented by graph coloring and a boundary volume hierarchy, respectively. However, the resulting booleans are not robust due to the meshing algorithms used to connect the partial meshes in the prototype. Therefore, further research is required to achieve a robust modeling workflow.

7.1.2. Simulation

There a several ways in which the efficient FEM system matrix assembly method described in Chapter 4 could be extended or applied to other problems. In Section 4.3, we noted that higher-order elements that omit cell nodes (for $p \geq 4$) or cell and face nodes (for $p \geq 3$) are also used in practice, next to the complete higher-order elements. Extension to such elements should be straightforward, by adapting Eq. (4.6) accordingly. For purely quadrilateral or hexahedral meshes, or their n-dimensional extensions, similar equations could be derived as well. However, a different approach, such as extracting regular parts of the mesh and treating those as structured grids, is likely to be even more efficient. Additionally, while we avoided a binning approach to maintain a simple implementation, further performance gains may be possible, as registers and shared memory could be used more effectively. However, potential gains are limited by the allocation times which currently amount to up to 47% of total assembly time and form a strict upper bound to potential improvements in computation time.

For adaptive simulations, local updates of the matrix may be faster than recreating the entire matrix. However, our approach is orthogonal to local updates, as local updates can benefit from exact allocation as well. Combining the two is a potential area for future research. Similarly, the approach could be applied to the acceleration of local, per-domain matrix assembly in distributed or multi-GPU simulations using domain decomposition.

An interesting future area of application for our approach is co-dimensional simulation on simplicial complexes, as pioneered by Zhu et al. [ZQC+14; ZLQF15] and recently adapted to elastic deformable objects by Chang et al. [CDGB19]. Although we only consider homogeneous simplicial complexes in Chapter 4, Eq. (4.6) applies to inhomogeneous simplicial complexes that have a mix of points, lines, triangles, and tetrahedra as top level elements as well. Several steps of the assembly process would require significant modifications. For example, in a co-dimensional setting triangles have element matrices as well that must be treated separately.

Another important application area for fast simulation methods is optimization. In the case of topology optimization using adaptive meshes, such as the moving mesh level set method by Liu and Korvink [LK08], our novel fast assembly method could potentially enable new methods that not only modify vertex positions, but also mesh connectivity adaptively.

7.1.3. Visualization

Several potential extensions to the remote visualization query compiler and streaming protocol introduced in Chapter 6 could be implemented to improve performance further or increase flexibility. Compression algorithms including those presented by O'Neil and Burtscher [OB11], Lindstrom [Lin14], or

Tao et al. [TDCC17] can be added to further reduce bandwidth requirements at the cost of additional processing on both client and server.

Queries could be extended to support subfields, i.e., named boundaries or subdomains, for instance an inlet in a computational fluid dynamics (CFD) simulation or a specific component in a computational structural mechanics (CSM) simulation. Especially in combination with reductions, for example averages or maximums of fields, such subfield queries could become useful. However, parallel reductions as required by the GPU back-end require reimplementation of many scalar optimizations such as common subexpression elimination, as parallelism cannot be expressed directly in LLVM-IR. Expressing parallelism in LLVM is a topic of ongoing research (see, e.g., [KJI+15]). Recently, Lattner and Pienaar have presented MLIR [LP19], an extensible multi-level intermediate representation designed for successive lowering to LLVM-IR with reusable optimization passes, which could provide an alternative solution as soon as it is made open source.

Furthermore, extensions to the type system could improve usability. For example, explicit vector types (instead of separate `VectorX` and `VectorY` fields) or calculations involving matrices and tensors would be useful for several physical domains, including CSM. Fields could also be annotated with physical units to detect mistakes due to adding fields with mismatched units.

Considering the unsatisfactory JavaScript deserialization performance (see Section 6.4.2), alternative serialization formats promising lower deserialization costs such as Cap'n Proto [San16] or FlatBuffers [Goo16], or JavaScript's native JavaScript object notation (JSON) format could be investigated. However, these typically come at an increased bandwidth cost. Alternatively, the use of Emscripten [Zak11] to target asm.js [HWZ14], an efficiently optimizable subset of JavaScript, or WebAssembly [Web19], a new portable binary code format for the web, may improve performance.

In a recent paper [KLN18], Kohn et al. have presented an adaptive execution model to accelerate databases that use compiled queries. By adaptively choosing between interpreted LLVM bytecode, unoptimized JIT-compiled code, and optimized JIT-compiled code, they reduce query response latencies. A similar approach could be used to reduce time to first image when using our remote visualization query compiler. However, direct interpretation of LLVM bytecode is currently only possible on the CPU. A custom GPU-accelerated interpreter could be used instead to hide compilation times.

To further reduce overhead when a small percentage of pixels are valid, our rixel encoding could be improved by applying run length encoding to the bitmap. However, frame coverages of less than 1.4% are not a core use case and applying run length encoding would increase deserialization cost.

7.1.4. Sparse Matrix Data Structures

With respect to GPU-optimized sparse matrix data structures, specifically the joint schedule and layout autotuning approach presented in Chapter 5, the extension to a more complete set of linear algebra operations would be beneficial. The `axpby` ($\mathbf{y} \leftarrow a\mathbf{x} + b\mathbf{y}$) and `dot` procedures are a good choice, as they would allow for implementation of several iterative solvers such as the conjugate gradient solver. Aside from the reduction within the dot product, however, they are trivially parallelizable and perform significantly fewer

operations than the matrix-vector product. Therefore, there is a lesser need for tuning, avoiding the need for weighting noted in the previous section.

Furthermore, while the Cartesian product approach to autotuning guarantees that the best variant is found, it is very expensive. By collecting a larger set of input matrices of varied structure, potentially by reusing the sparsity structure but not the entries of matrices in existing matrix collections, a predictive tuning model could be built. This could potentially be achieved using machine learning to estimate layout and schedule parameters (see, e.g., [AKC+19]).

An extension to full-fledged sparse tensor algebras as TACO offers for CPU codes [KKC+17] involves many interesting challenges. In particular, can the padded transposed and sliced layouts be generalized to tensors? How would they interact with hypersparse (low-rank) matrices or tensors? Slicing could potentially be represented as an index transform, i.e., $y_i = A_{ij}x_j \rightarrow y_i = A'_{\lfloor i/s \rfloor (i \bmod s)j}x_j$, where s is the slice size.

Another extension would be the support of encoded entries. For example, Zayer et al. [ZSS17a] and the TCSR approach described in Chapter 3 each use sparse matrices to describe meshes. The former encode integer values in the order of entries of a compressed sparse column (CSC) matrix, while the latter encode the sign of a ternary matrix in the column index of CSR matrices. Compound entries can also benefit from compact encodings. For example, consider the unit quaternion matrices used in various methods (see, e.g., [CKPS18]), which can be encoded using only three values and a sign bit, reducing memory and bandwidth requirements by nearly 25%. However, the advantage of alignment for single precision quaternions would be lost.

Bibliography

[ADSF16] Altenhofen, C., A. Dietrich, A. Stork, and D. W. Fellner. "Rixels: Towards Secure Interactive 3D Graphics in Engineering Clouds." In: *Transactions on Internet Research* 12(1) (2016), pp. 31–38. ISSN: 1820-4503 (cit. on pp. 5, 75, 76, 79, 80, 86, 92, 93).

[ALM+17] Altenhofen, C., F. Loosmann, J. S. Mueller-Roemer, T. Grasser, T. H. Luu, and A. Stork. "Integrating Interactive Design and Simulation for Mass Customized 3D-Printed Objects – A Cup Holder Example." In: *Proceedings of the 28th Annual International Solid Freeform Fabrication Symposium – An Additive Manufacturing Conference*. SFF '17. 2017, pp. 2289–2301 (cit. on pp. 1, 4).

[ASSF17] Altenhofen, C., F. Schuwirth, A. Stork, and D. W. Fellner. "Implicit Mesh Generation using Volumetric Subdivision." In: *Workshop on Virtual Reality Interaction and Physical Simulation*. VRIPHYS '17. 2017. DOI: 10.2312/vriphys.20171079 (cit. on pp. 2, 21, 24, 43).

[AJ05] Alumbaugh, T. J. and X. Jiao. "Compact Array-Based Mesh Data Structures." In: *Proceedings of the 14th International Meshing Roundtable*. 2005, pp. 485–503. DOI: 10.1007/3-540-29090-7_29 (cit. on p. 22).

[ADFQ17] Anzt, H., J. Dongarra, G. Flegar, and E. S. Quintana-Ortí. "Batched Gauss-Jordan Elimination for Block-Jacobi Preconditioner Generation on GPUs." In: *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM '17. 2017, pp. 1–10. DOI: 10.1145/3026937.3026940 (cit. on p. 52).

[ATD14] Anzt, H., S. Tomov, and J. Dongarra. *Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs*. Tech. rep. ut-eecs-14-727. University of Tennessee, 2014 (cit. on p. 44).

[AB03] Ascher, U. M. and E. Boxerman. "On the modified conjugate gradient method in cloth simulation." In: *The Visual Computer* 19(7-8) (2003), pp. 526–531. DOI: 10.1007/s00371-003-0220-4 (cit. on p. 52).

[AKC+19] Ashouri, A. H., W. Killian, J. Cavazos, G. Palermo, and C. Silvano. "A Survey on Compiler Autotuning using Machine Learning." In: *ACM Computing Surveys* 51(5) (2019), 13:1–13:42. DOI: 10.1145/3197978 (cit. on p. 101).

[Aut18] Autodesk. *Meshmixer*. Comp. software. Version 3.5. 2018 (cit. on pp. 2, 4).

[BG92] Babuška, I. and B. Q. Guo. "The *h*, *p* and *h-p* Version of the Finite Element Method: Basis Theory and Applications." In: *Advances in Engineering Software* 15(3–4) (1992), pp. 159–174. DOI: 10.1016/0965-9978(92)90097-Y (cit. on p. 58).

[BW98] Baraff, D. and A. Witkin. “Large Steps in Cloth Simulation.” In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. SIGGRAPH ’98. 1998, pp. 43–54. DOI: 10.1145/280814.280821 (cit. on p. 52).

[BG08] Bell, N. and M. Garland. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation, 2008 (cit. on p. 11).

[BH15] Bell, N. and J. Hoberock. *Thrust 1.8.1*. 2015. URL: https://thrust.github.io/ (cit. on pp. 28, 34, 52, 87).

[BD13] Bender, J. and C. Deul. “Adaptive Cloth Simulation using Corotational Finite Elements.” In: *Computers & Graphics* 37(7) (2013), pp. 820–829. DOI: 10.1016/j.cag.2013.04.008 (cit. on p. 42).

[BMFG18] Biedert, T., P. Messmer, T. Fogal, and C. Garth. “Hardware-Accelerated Multi-Tile Streaming for Realtime Remote Visualization.” In: *Eurographics Symposium on Parallel Graphics and Visualization*. EGPGV ’18. 2018. DOI: 10.2312/pgv.20181093 (cit. on pp. 76, 79).

[Bik96] Bik, A. J. C. “Compiler Support for Sparse Matrix Computations.” PhD thesis. Rijksuniversiteit Leiden, 1996 (cit. on p. 65).

[BGM19] Bormann, P., R. Gutbell, and J. S. Mueller-Roemer. “Integrating Server-based Simulations into Web-based Geo-applications.” In: *Eurographics 2019 - Short Papers*. 2019. DOI: 10.2312/egs.20191012 (cit. on pp. 75, 86).

[Bri08] Bridson, R. *Fluid Simulation For Computer Graphics*. Ak Peters Series. A K Peters, Limited, 2008. ISBN: 1-56881-326-0 (cit. on p. 81).

[BSA12] Brodtkorb, A. R., M. L. Sætra, and M. Altinakar. “Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation.” In: *Computers & Fluids* 55 (2012), pp. 1–12. DOI: 10.1016/j.compfluid.2011.10.012 (cit. on p. 87).

[BHU10a] Burkhart, D., B. Hamann, and G. Umlauf. “Adaptive and Feature-Preserving Subdivision for High-Quality Tetrahedral Meshes.” In: *Computer Graphics Forum* 29(1) (2010), pp. 117–127. DOI: 10.1111/j.1467-8659.2009.01581.x (cit. on p. 24).

[BHU10b] Burkhart, D., B. Hamann, and G. Umlauf. “Iso-geometric Finite Element Analysis Based on Catmull-Clark Subdivision Solids.” In: *Computer Graphics Forum* 29(5) (Symposium on Geometry Processing 2010), pp. 1575–1584. DOI: 10.1111/j.1467-8659.2010.01766.x (cit. on pp. 21, 30).

[CNGT14] Cao, T.-T., A. Nanjappa, M. Gao, and T.-S. Tan. “A GPU accelerated algorithm for 3D Delaunay triangulation.” In: *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’14. 2014. DOI: 10.1145/2556700.2556710 (cit. on p. 2).

[CC78] Catmull, E. and J. Clark.
"Recursively generated B-spline surfaces on arbitrary topological meshes."
In: *Computer-Aided Design* 10(6) (1978), pp. 350–355.
DOI: 10.1016/0010-4485(78)90110-0 (cit. on pp. 24, 30).

[CLD10] Cecka, C., A. J. Lew, and E. Darve.
"Assembly of Finite Element Methods on Graphics Processors."
In: *International Journal for Numerical Methods in Engineering* 85(5) (2010), pp. 640–669.
DOI: 10.1002/nme.2989 (cit. on p. 45).

[CGA18] CGAL. *Computational Geometry Algorithms Library*. 2018. URL: https://www.cgal.org (cit. on p. 70).

[CGN18] CGNS Steering Committee. *CGNS Standard Interface Data Structures 3.3 (rev 2)*. 2018.
URL: https://cgns.github.io/CGNS_docs_current/sids/sids.pdf (visited on 2018-06-08) (cit. on p. 47).

[CDGB19] Chang, J., F. Da, E. Grinspun, and C. Batty. "A Unified Simplicial Model for Mixed-Dimensional and Non-Manifold Deformable Elastic Objects." In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 2(2) (2019), 11:1–11:18.
DOI: 10.1145/3340252 (cit. on p. 99).

[CMQ03] Chang, Y.-S., K. T. McDonnell, and H. Qin.
"An interpolatory subdivision for volumetric models over simplicial complexes."
In: *2003 Shape Modeling International*. IEEE. 2003, pp. 143–152.
DOI: 10.1109/smi.2003.1199610 (cit. on p. 24).

[CCT+11] Chen, K.-T., Y.-C. Chang, P.-H. Tseng, C.-Y. Huang, and C.-L. Lei.
"Measuring the Latency of Cloud Gaming Systems."
In: *Proceedings of the 19th ACM International Conference on Multimedia*. MM '11. 2011, pp. 1269–1272. DOI: 10.1145/2072298.2071991 (cit. on p. 79).

[CC17] Chen, S. C. and W. C. Chew.
"Numerical electromagnetic frequency domain analysis with discrete exterior calculus."
In: *Journal of Computational Physics* 350 (2017), pp. 668–689.
DOI: 10.1016/j.jcp.2017.08.068 (cit. on p. 25).

[CSY+15] Cheng, Z., E. Shaffer, R. Yeh, G. Zagaris, and L. Olson.
"Efficient parallel optimization of volume meshes on heterogeneous computing systems."
In: *Engineering with Computers* (2015). DOI: 10.1007/s00366-014-0393-7
(cit. on p. 23).

[CKPS18] Chern, A., F. Knöppel, U. Pinkall, and P. Schröder. "Shape from Metric."
In: *ACM Transactions on Graphics* 37(4) (2018), 63:1–63:17.
DOI: 10.1145/3197517.3201276 (cit. on pp. 62, 64, 101).

[CKSD17] Cheshmi, K., S. Kamil, M. M. Strout, and M. M. Dehnavi.
"Sympiler: transforming sparse matrix codes by decoupling symbolic analysis."
In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. 2017. DOI: 10.1145/3126908.3126936 (cit. on pp. 65, 73).

[CKR+12] Chiw, C., G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. "Diderot: A Parallel DSL for Image Analysis and Visualization." In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. 2012, pp. 111–120. DOI: 10.1145/2254064.2254079 (cit. on p. 78).

[CCQ+14] Choi, H., W. Choi, T. M. Quan, D. Hildebrand, H. Pfister, and W.-K. Jeong. "Vivaldi: A Domain-Specific Language for Volume Processing and Visualization on Distributed Heterogeneous Systems." In: *Visualization and Computer Graphics, IEEE Transactions on* 20(12) (2014), pp. 2407–2416. DOI: 10.1109/TVCG.2014.2346322 (cit. on p. 78).

[CSV10] Choi, J. W., A. Singh, and R. W. Vuduc. "Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs." In: *ACM SIGPLAN Notices* 45(5) (2010), pp. 115–126. DOI: 10.1145/1837853.1693471 (cit. on p. 44).

[Cho17] Choquette, J. "NVIDIA's Volta GPU: Programmability and Performance for GPU Computing." In: *Hot Chips: A Symposium on High Performance Chips*. HC29. 2017 (cit. on p. 9).

[Cox73] Coxeter, H. S. M. *Regular Polytopes*. 3rd ed. Dover Publications Inc., 1973. ISBN: 0-486-61480-8 (cit. on p. 24).

[CPS11] Crane, K., U. Pinkall, and P. Schröder. "Spin transformations of discrete surfaces." In: *ACM Transactions on Graphics* 30(4) (2011), 104:1–104:10. DOI: 10.1145/2010324.1964999 (cit. on pp. 63, 68).

[CPS13] Crane, K., U. Pinkall, and P. Schröder. "Robust Fairing via Conformal Curvature Flow." In: *ACM Transactions on Graphics* 32(4) (2013), 61:1–61:10. DOI: 10.1145/2461912.2461986 (cit. on p. 63).

[CJS15] Cuvelier, F., C. Japhet, and G. Scarella. "An efficient way to assemble finite element matrices in vector languages." In: *BIT Numerical Mathematics* 56(3) (2015), pp. 833–864. DOI: 10.1007/s10543-015-0587-4 (cit. on p. 45).

[DV13] D'Amato, J. P. and M. Vénere. "A CPU-GPU Framework for Optimizing the Quality of Large Meshes." In: *Journal of Parallel and Distributed Computing* 73(8) (2013), pp. 1127–1134. DOI: 10.1016/j.jpdc.2013.03.007 (cit. on p. 23).

[Dam16] Damiand, G. *CGAL 4.9 User Manual*. ch. Linear Cell Complex. 2016. URL: http://doc.cgal.org/latest/Linear_cell_complex/index.html (cit. on p. 21).

[Dav18] Davis, T. *SuiteSparse: A suite of sparse matrix packages*. 2018. URL: http://faculty.cse.tamu.edu/davis/suitesparse.html (visited on 2018-06-08) (cit. on p. 56).

[DH11] Davis, T. A. and Y. Hu. "The University of Florida Sparse Matrix Collection." In: *ACM Transactions on Mathematical Software* 38(1) (2011), 1:1–1:25. DOI: 10.1145/2049662.2049663 (cit. on p. 68).

[DGH04] De Floriani, L., D. Greenfieldboyce, and A. Hui.
"A data structure for non-manifold simplicial d-complexes." In: *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*. SGP '04. 2004, pp. 83–92. DOI: 10.1145/1057432.1057444 (cit. on p. 22).

[DH03] De Floriani, L. and A. Hui.
"A scalable data structure for three-dimensional non-manifold objects." In: *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*. SGP '03. 2003. ISBN: 1-58113-687-0 (cit. on p. 22).

[DT07] DeCoro, C. and N. Tatarchuk. "Real-time Mesh Simplification Using the GPU."
In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. I3D '07. 2007. DOI: 10.1145/1230100.1230128 (cit. on p. 23).

[DCG18] Delaunay, X., A. Courtois, and F. Gouillon. "Evaluation of lossless and lossy algorithms for the compression of scientific datasets in NetCDF-4 or HDF5 formatted files."
In: *Geoscientific Model Development Discussions* (2018), pp. 1–25.
DOI: 10.5194/gmd-2018-250. In review (cit. on p. 79).

[DZSS17] Derler, A., R. Zayer, H.-P. Seidel, and M. Steinberger.
"Dynamic scheduling for efficient hierarchical sparse matrix operations on the GPU."
In: *Proceedings of the International Conference on Supercomputing*. ICS '17. 2017.
DOI: 10.1145/3079079.3079085 (cit. on p. 66).

[DKT06] Desbrun, M., E. Kanso, and Y. Tong.
"Discrete Differential Forms for Computational Modeling."
In: *ACM SIGGRAPH 2006 Courses*. 2006, pp. 39–54. DOI: 10.1145/1185657.1185665 (cit. on p. 24).

[DS19] Deveria, A. and L. Schoors. *Can I use*. 2019.
URL: https://caniuse.com/ (visited on 2019-02-19) (cit. on p. 82).

[DE12] Di Pietro, D. A. and A. Ern. *Mathematical Aspects of Discontinuous Galerkin Methods*.
Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-22980-0 (cit. on p. 47).

[DPS14] DiCarlo, A., A. Paoluzzi, and V. Shapiro.
"Linear algebraic representation for topological structures."
In: *Computer-Aided Design* 46 (2014), pp. 269–274. DOI: 10.1016/j.cad.2013.08.044 (cit. on p. 22).

[DBWR09] Duke, D., R. Borgo, M. Wallace, and C. Runciman. "Huge Data But Small Programs: Visualization Design via Multiple Embedded DSLs." English.
In: *Practical Aspects of Declarative Languages*. Vol. 5418.
Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 31–45.
DOI: 10.1007/978-3-540-92995-6_3 (cit. on p. 78).

[DRE+15] Dyedov, V., N. Ray, D. Einstein, X. Jiao, and T. J. Tautges. "AHF: Array-Based Half-Facet Data Structure for Mixed-Dimensional and Non-manifold Meshes."
In: *Engineering with Computers* 31 (2015), pp. 389–404.
DOI: 10.1007/s00366-014-0378-6 (cit. on p. 22).

[DLM11] Dziekonski, A., A. Lamecki, and M. Mrozowski.
"A Memory Efficient and fast Sparse Matrix Vector Product on a GPU."
In: *Progress In Electromagnetics Research* 116 (2011), pp. 49–63.
DOI: 10.2528/PIER11031607 (cit. on p. 44).

[EGH00] Eymard, R., T. Gallouët, and R. Herbin. "Finite volume methods."
In: *Handbook of Numerical Analysis*. Vol. 7. 2000, pp. 713–1018.
DOI: 10.1016/S1570-8659(00)07005-8 (cit. on p. 15).

[FAB+18] Feng, L., P. Alliez, L. Busé, H. Delingette, and M. Desbrun.
"Curved Optimal Delaunay Triangulation."
In: *ACM Transactions on Graphics* 37(4) (2018), 61:1–61:16.
DOI: 10.1145/3197517.3201358 (cit. on p. 46).

[FS19] Feng, W. and T. Scogland. *Green500 List*. June 2019.
URL: https://www.top500.org/green500/list/2019/06/ (cit. on p. 1).

[FM96] Foster, N. and D. Metaxas. "Realistic animation of liquids."
In: *Graphical models and image processing* 58(5) (1996), pp. 471–483 (cit. on p. 81).

[GR09] Geuzaine, C. and J.-F. Remacle. "Gmsh: A 3-D Finite Element Mesh Generator with Built-in Pre- and Post-processing Facilities." In: *International Journal for Numerical Methods in Engineering* 79(11) (2009), pp. 1309–1331. DOI: 10.1002/nme.2579 (cit. on p. 70).

[Goo08] Google. *Protocol Buffers (protobuf)*. 2008.
URL: https://github.com/google/protobuf (visited on 2016-03-14) (cit. on p. 82).

[Goo16] Google. *Flatbuffers*. 2016.
URL: https://github.com/google/flatbuffers (visited on 2016-03-14)
(cit. on p. 100).

[Gri13] Grigorik, I. *High Performance Browser Networking*. O'Reilly Media, 2013.
ISBN: 978-1-4493-4476-4 (cit. on p. 79).

[GLG+15] Guo, X., M. Lange, G. Gorman, L. Mitchell, and M. Weiland.
"Developing a Scalable Hybrid MPI/OpenMP Unstructured Finite Element Model."
In: *Computers & Fluids* 110 (2015), pp. 227–234.
DOI: 10.1016/j.compfluid.2014.09.007 (cit. on pp. 4, 42).

[HW65] Harlow, F. H. and J. E. Welch. "Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with a Free Surface."
In: *The Physics of Fluids* 8(12) (1965), pp. 2182–2189 (cit. on p. 81).

[Hat02] Hatcher, A. *Algebraic Topology*. Cambridge University Pr., 2002. ISBN: 0-521-79540-0
(cit. on pp. 24, 33, 95).

[HWZ14] Herman, D., L. Wagner, and A. Zakai. *asm.js*. Specification. 2014.
URL: http://asmjs.org/spec/latest/ (cit. on p. 100).

[HL90] Hile, G. N. and P. Lounesto. "Matrix Representations of Clifford Algebras."
In: *Linear Algebra and its Applications* 128 (1990), pp. 51–63.
DOI: 10.1016/0024-3795(90)90282-H (cit. on p. 65).

[HKL+99] Hoff, K. E., J. Keyser, M. Lin, D. Manocha, and T. Culver.
"Fast computation of generalized Voronoi diagrams using graphics hardware." In:
Proceedings of the 26th annual conference on Computer graphics and interactive techniques.
SIGGRAPH '99. 1999. DOI: 10.1145/311535.311567 (cit. on p. 11).

[HHE15] Holmberg, C., S. Hakansson, and G. Eriksson.
Web Real-Time Communication Use Cases and Requirements. RFC 7478. 2015.
DOI: 10.17487/RFC7478 (cit. on p. 82).

[HZG+18] Hu, Y., Q. Zhou, X. Gao, A. Jacobson, D. Zorin, and D. Panozzo.
"Tetrahedral Meshing in the Wild."
In: *ACM Transactions on Graphics* 37(4) (2018), 60:1–60:14.
DOI: 10.1145/3197517.3201353 (cit. on pp. 2, 43).

[Int18] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. 325462-068US. Manual. 2018
(cit. on pp. 10, 11).

[ISF07] Irving, G., C. Schroeder, and R. Fedkiw.
"Volume Conserving Finite Element Simulations of Deformable Models."
In: *ACM Transactions on Graphics* 26(3) (2007), 13:1–13:6.
DOI: 10.1145/1276377.1276394 (cit. on p. 43).

[Jin14] Jin, J. *The Finite Element Method in Electromagnetics*. Third. Wiley-IEEE Press, 2014.
ISBN: 1118571363 (cit. on p. 63).

[JM96] Joy, K. I. and R. MacCracken. *The Refinement Rules for Catmull-Clark Solids*.
Tech. rep. CSE-96-1. University of California, Davis, 1996 (cit. on pp. 21, 24, 30).

[KCŽO07] Kavan, L., S. Collins, J. Žára, and C. O'Sullivan. "Skinning with Dual Quaternions."
In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. I3D '07.
2007, pp. 39–46. DOI: 10.1145/1230100.1230107 (cit. on p. 62).

[KAB+16] Kepner, J. et al. "Mathematical foundations of the GraphBLAS."
In: *2016 IEEE High Performance Extreme Computing Conference*. 2016.
DOI: 10.1109/hpec.2016.7761646 (cit. on pp. 29, 64).

[KO18] Kessenich, J. and B. Ouriel. *SPIR-V Specification*. Version 1.00, Rev. 12.
Khronos Group, 2018.
URL: https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf
(cit. on p. 84).

[KJI+15] Khaldi, D., P. Jouvelot, F. Irigoin, C. Ancourt, and B. Chapman.
"LLVM Parallel Intermediate Representation: Design and Evaluation using OpenSHMEM Communications."
In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*.
LLVM '15. ACM, 2015, 2:1–2:8. DOI: 10.1145/2833157.2833158 (cit. on p. 100).

[Khr18] Khronos OpenCL Working Group. *The OpenCL™ Specification*. Version 2.2-8. 2018.
URL: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf
(cit. on pp. 11, 14).

[KY88] Kincaid, D. R. and D. M. Young. “A Brief Review of the ITPACK Project.” In: *Journal of Computational and Applied Mathematics* 24(1–2) (1988), pp. 121–127. DOI: 10.1016/0377-0427(88)90347-0 (cit. on p. 16).

[KCS+16] Kindlmann, G., C. Chiw, N. Seltzer, L. Samuels, and J. Reppy. “Diderot: a Domain-Specific Language for Portable Parallel Scientific Visualization and Image Analysis.” In: *IEEE Transactions on Visualization and Computer Graphics* 22(1) (2016), pp. 867–876. DOI: 10.1109/tvcg.2015.2467449 (cit. on p. 78).

[KGKM16] King, J., T. Gilray, R. M. Kirby, and M. Might. “Dynamic Sparse-Matrix Allocation on GPUs.” In: *High Performance Computing. ISC High Performance 2016*. Vol. 9697. Lecture Notes in Computer Science. 2016, pp. 61–80. DOI: 10.1007/978-3-319-41321-1_4 (cit. on p. 98).

[KKC+17] Kjolstad, F., S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. “The Tensor Algebra Compiler.” In: *Proceedings of the ACM on Programming Languages* 1(OOPSLA) (2017), 77:1–77:29. DOI: 10.1145/3133901 (cit. on pp. 65, 101).

[KLN18] Kohn, A., V. Leis, and T. Neumann. “Adaptive Execution of Compiled Queries.” In: *2018 IEEE 34th International Conference on Data Engineering*. ICDE ’18. 2018. DOI: 10.1109/icde.2018.00027 (cit. on p. 100).

[KME09] Komatitsch, D., D. Michéa, and G. Erlebacher. “Porting a High-Order Finite-Element Earthquake Modeling Application to NVIDIA Graphics Cards using CUDA.” In: *Journal of Parallel and Distributed Computing* 69(5) (2009), pp. 451–460. DOI: 10.1016/j.jpdc.2009.01.006 (cit. on p. 45).

[KLB14] Koschier, D., S. Lipponer, and J. Bender. “Adaptive Tetrahedral Meshes for Brittle Fracture Simulation.” In: *Proceedings of the 2014 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA ’14. 2014, pp. 57–66 (cit. on p. 42).

[Kra17] Kranz, J. *Methodik und Richtlinien für die Konstruktion von laseradditiv gefertigten Leichtbaustrukturen*. Light Engineering für die Praxis. Springer Vieweg, 2017. DOI: 10.1007/978-3-662-55339-8 (cit. on pp. 1, 41, 42).

[KBK13] Kremer, M., D. Bommes, and L. Kobbelt. “OpenVolumeMesh: A Versatile Index-Based Data Structure for 3D Polytopal Complexes.” In: *Proceedings of the 21st International Meshing Roundtable*. 2013, pp. 531–548. DOI: 10.1007/978-3-642-33573-0_31 (cit. on pp. 21, 22, 35).

[KHW+12] Kreutzer, M., G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop. “Sparse Matrix-vector Multiplication on GPGPU Clusters: A New Storage Format and a Scalable Implementation.” In: *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IPDPSW ’12. 2012, pp. 1696–1702. DOI: 10.1109/IPDPSW.2012.211 (cit. on p. 44).

[KW03] Krüger, J. and R. Westermann.
"Linear algebra operators for GPU implementation of numerical algorithms."
In: *ACM Transactions on Graphics* 22(3) (2003), pp. 908–916.
DOI: 10.1145/882262.882363 (cit. on p. 11).

[LaC01] LaCourse, D. "Virtual Prototyping Pays Off." In: *Cadalyst* (2001).
URL: http://www.cadalyst.com/manufacturing/virtual-prototyping-pays-9774
(visited on 2019-01-11) (cit. on p. 1).

[LLLV05] Lage, M., T. Lewiner, H. Lopes, and L. Velho.
"CHF: A Scalable Topological Data Structure for Tetrahedral Meshes."
In: *XVIII Brazilian Symposium on Computer Graphics and Image Processing*. SIBGRAPI '05.
2005. DOI: 10.1109/sibgrapi.2005.18 (cit. on p. 22).

[LLA11] Lang, H., J. Linn, and M. Arnold.
"Multi-body Dynamics Simulation of Geometrically Exact Cosserat Rods."
In: *Multibody System Dynamics* 25(3) (2011), pp. 285–312.
DOI: 10.1007/s11044-010-9223-x (cit. on p. 64).

[LM01] Larsen, E. S. and D. McAllister. "Fast matrix multiplies using graphics hardware."
In: *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. Supercomputing '01.
2001. DOI: 10.1145/582034.582089 (cit. on p. 11).

[LA04a] Lattner, C. and V. Adve.
"LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation."
In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. 2004, pp. 75–86 (cit. on p. 64).

[LA04b] Lattner, C. and V. Adve.
"LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation."
In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*.
2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665 (cit. on p. 77).

[LP19] Lattner, C. and J. Pienaar.
"MLIR Primer: A Compiler Infrastructure for the End of Moore's Law."
In: *Compilers for Machine Learning. C4ML workshop at CGO 2019*. 2019.
URL: https://www.c4ml.org/#h.p_Euf4yx0JmOIA (cit. on p. 100).

[LGMF17] Ledur, C., D. Griebler, I. Manssour, and L. G. Fernandes.
"A High-Level DSL for Geospatial Visualizations with Multi-core Parallelism Support."
In: *2017 IEEE 41st Annual Computer Software and Applications Conference*. COMPSAC '17.
2017. DOI: 10.1109/compsac.2017.18 (cit. on pp. 5, 78).

[Li16] Li, K. "OpenMP Accelerator Support for GPU." In: *OpenPOWER Summit*. 2016
(cit. on p. 12).

[LMG+18] Li, S., N. Marsaglia, C. Garth, J. Woodring, J. Clyne, and H. Childs.
"Data Reduction Techniques for Simulation, Visualization and Data Analysis."
In: *Computer Graphics Forum* 37(6) (2018), pp. 422–447. DOI: 10.1111/cgf.13336
(cit. on p. 79).

[Lin14] Lindstrom, P. “Fixed-Rate Compressed Floating-Point Arrays.” In: *Visualization and Computer Graphics, IEEE Transactions on* 20(12) (2014), pp. 2674–2683. DOI: 10.1109/TVCG.2014.2346458 (cit. on pp. 78, 99).

[LJC17] Liu, H.-T. D., A. Jacobson, and K. Crane. “A Dirac Operator for Extrinsic Shape Analysis.” In: *Computer Graphics Forum* 36(5) (Symposium on Geometry Processing 2017), pp. 139–149. DOI: 10.1111/cgf.13252 (cit. on p. 63).

[LV14] Liu, W. and B. Vinter. “An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data.” In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 370–381. DOI: 10.1109/IPDPS.2014.47 (cit. on p. 98).

[LV15] Liu, W. and B. Vinter. “A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors.” In: *Journal of Parallel and Distributed Computing* 85 (2015), pp. 47–61. DOI: 10.1016/j.jpdc.2015.06.010 (cit. on pp. 44, 45, 54, 58).

[LS15] Liu, Y. and B. Schmidt. “LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs.” In: *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors*. ASAP. 2015, pp. 82–89. DOI: 10.1109/ASAP.2015.7245713 (cit. on pp. 44, 98).

[LJWD08] Liu, Y., S. Jiao, W. Wu, and S. De. “GPU Accelerated Fast FEM Deformation Simulation.” In: *2008 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 2008, pp. 606–609. DOI: 10.1109/APCCAS.2008.4746096 (cit. on pp. 3, 4, 42).

[LK08] Liu, Z. and J. G. Korvink. “Adaptive moving mesh level set method for structure topology optimization.” In: *Engineering Optimization* 40(6) (2008), pp. 529–558. DOI: 10.1080/03052150801985544 (cit. on p. 99).

[LCA18] Livesu, M., D. Cabiddu, and M. Attene. “Slice2mesh: Meshing Sliced Data for the Simulation of AM Processes.” In: *Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference* (2018). DOI: 10.2312/stag.20181294 (cit. on p. 43).

[LLV19] LLVM Project. *Performance Tips for Frontend Authors*. 2019. URL: https://llvm.org/docs/Frontend/PerformanceTips.html (visited on 2019-02-19) (cit. on p. 84).

[Loo87] Loop, C. T. “Smooth Subdivision Surfaces Based on Triangles.” MSc Thesis. University of Utah, 1987 (cit. on p. 24).

[Lot18] Lotter, L. “GPU-basierte Shallow Water Simulation.” BSc Thesis. Technische Universität Darmstadt, 2018 (cit. on p. 87).

[Ma05] Ma, W. “Subdivision surfaces for CAD—an overview.” In: *Computer-Aided Design* 37(7) (2005), pp. 693–709. DOI: 10.1016/j.cad.2004.08.008 (cit. on p. 23).

[MS17] Mantor, M. and B. Sander. “AMD’s Radeon Next Generation GPU Architecture. Vega10.” In: *Hot Chips: A Symposium on High Performance Chips*. HC29. 2017 (cit. on p. 9).

[Map16] MapD Technologies, Inc. *MapD*. 2016.
URL: http://www.mapd.com/ (visited on 2016-03-14) (cit. on p. 78).

[MO07] Masuda, H. and K. Ogawa.
"Application of Interactive Deformation to Assembled Mesh Models for CAE Analysis." In: *Proceedings of the ASME 2007 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. 33rd Design Automation Conference*. Vol. 6.A and B. 2007, pp. 469–477. DOI: 10.1115/detc2007-34636 (cit. on p. 2).

[MBB+13] Mattson, T., D. Bader, J. Berry, et al. "Standards for Graph Algorithm Primitives."
In: *2013 IEEE High Performance Extreme Computing Conference*. 2013.
DOI: 10.1109/hpec.2013.6670338 (cit. on pp. 15, 28, 95, 98).

[MB16] Meder, J. and B. Brüderlin.
"Fast Depth Image Based Rendering for synthetic frame extrapolation."
In: *Journal of Theoretical and Applied Computer Science* 10(3) (2016), pp. 3–18.
ISSN: 2299-2634 (cit. on p. 87).

[MF11] Melnikov, A. and I. Fette. *The WebSocket Protocol*. RFC 6455. 2011.
DOI: 10.17487/RFC6455 (cit. on p. 82).

[MLP+17] Memeti, S., L. Li, S. Pllana, J. Kołodziej, and C. Kessler.
"Benchmarking OpenCL, OpenACC, OpenMP, and CUDA." In: *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*.
ARMS-CC '17. 2017. DOI: 10.1145/3110355.3110356 (cit. on p. 12).

[Mer18] Merill, D. *CUB 1.8.0*. 2018.
URL: https://github.com/NVlabs/cub/releases/tag/v1.8.0 (cit. on pp. 34, 52).

[Mic16] Microsoft. *Remote Desktop Protocol*. 2016.
URL: https://msdn.microsoft.com/en-us/library/aa383015(VS.85).aspx (visited on 2016-03-14) (cit. on p. 79).

[Mic19] Microsoft. *Direct3D 11 Graphics*. 2019.
URL: https://docs.microsoft.com/windows/desktop/direct3d11/ (cit. on p. 11).

[MWS+19] Mlakar, D., M. Winter, H.-P. Seidel, M. Steinberger, and R. Zayer.
AlSub: Fully Parallel and Modular Subdivision. Preprint. Jan. 16, 2019.
arXiv: 1809.06047v3 [cs.GR] (cit. on p. 23).

[MLA10] Monakov, A., A. Lokhmotov, and A. Avetisyan.
"Automatically Tuning Sparse Matrix-vector Multiplication for GPU Architectures."
In: *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*. HiPEAC'10. 2010. DOI: 10.1007/978-3-642-11515-8_10
(cit. on pp. 63, 65, 66).

[Mor18] Morgan, T. P. *The End Of Xeon Phi – It's Xeon And Maybe GPUs From Here*. 2018.
URL: https://www.nextplatform.com/2018/07/27/end-of-the-line-for-xeon-phi-its-all-xeon-from-here/ (cit. on p. 9).

[Mos18] Mostak, T. “Speed at Scale: Using GPUs to Accelerate Analytics for Extreme Use Cases.” In: *NVIDIA GPU Technology Conference*. 2018. URL: http://on-demand.gputechconf.com/gtc/2018/presentation/s81008-speed-at-scale-using-gpus-to-accelerate-analytics-for-extreme-use-cases-presented-by-mapd.pdf (cit. on p. 78).

[MŞ15] Mostak, T. and A. Şuhan. *MapD: Massive Throughput Database Queries with LLVM on GPUs*. 2015. URL: https://devblogs.nvidia.com/parallelforall/mapd-massive-throughput-database-queries-llvm-gpus (visited on 2016-03-14) (cit. on pp. 5, 78).

[MA16] Mueller-Roemer, J. S. and C. Altenhofen. “JIT-compilation for Interactive Scientific Visualization.” In: *Short Papers Proceedings: 24th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*. WSCG ’16. 2016, pp. 197–206 (cit. on pp. 64, 75, 96).

[MAS17] Mueller-Roemer, J. S., C. Altenhofen, and A. Stork. “Ternary Sparse Matrix Representation for Volumetric Mesh Subdivision and Processing on GPUs.” In: *Computer Graphics Forum* 36(5) (Symposium on Geometry Processing 2017), pp. 59–69. DOI: 10.1111/cgf.13245 (cit. on pp. 19, 95, 96).

[MS18] Mueller-Roemer, J. S. and A. Stork. “GPU-based Polynomial Finite Element Matrix Assembly for Simplex Meshes.” In: *Computer Graphics Forum* 37(7) (Pacific Graphics 2018), pp. 443–454. DOI: 10.1111/cgf.13581 (cit. on pp. 19, 64, 96).

[MSF19] Mueller-Roemer, J. S., A. Stork, and D. W. Fellner. “Joint Schedule and Layout Autotuning for Sparse Matrices with Compound Entries on GPUs.” In: *Vision, Modeling and Visualization*. VMV ’19. 2019, pp. 109–116. DOI: 10.2312/vmv.20191324 (cit. on p. 96).

[MAS+16] Mullapudi, R. T., A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. “Automatically Scheduling Halide Image Processing Pipelines.” In: *ACM Transactions on Graphics* 35(4) (2016), 83:1–83:11. DOI: 10.1145/2897824.2925952 (cit. on p. 64).

[MP78] Muller, D. E. and F. P. Preparata. “Finding the intersection of two convex polyhedra.” In: *Theoretical Computer Science* 7(2) (1978), pp. 217–236. DOI: 10.1016/0304-3975(78)90051-8 (cit. on p. 21).

[MG04] Müller, M. and M. Gross. “Interactive Virtual Materials.” In: *Proceedings of Graphics Interface 2004*. GI ’04. 2004, pp. 239–246 (cit. on p. 43).

[Nan12] Nanjappa, A. “Delaunay Triangulation in R^3 on the GPU.” PhD thesis. National University of Singapore, 2012 (cit. on p. 2).

[NMG09] Ng, Y. T., C. Min, and F. Gibou. “An efficient fluid-solid coupling algorithm for single-phase flows.” In: *Journal of Computational Physics* 228(23) (2009), pp. 8807–8829. DOI: 10.1016/j.jcp.2009.08.032 (cit. on p. 81).

[NBGS08] Nickolls, J., I. Buck, M. Garland, and K. Skadron. "Scalable Parallel Programming with CUDA." In: *ACM Queue* 6(2) (2008), pp. 40–53. DOI: 10.1145/1365490.1365500 (cit. on p. 80).

[NVI16] NVIDIA Corporation. *NVIDIA GeForce GTX 1080. Gaming Perfected*. Whitepaper. 2016. URL: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf (cit. on p. 9).

[NVI17] NVIDIA Corporation. *NVIDIA Tesla P100*. Whitepaper. 2017. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (cit. on p. 10).

[NVI18a] NVIDIA Corporation. *CUDA C Programming Guide*. PG-02829-001_v10.0. 2018. URL: https://docs.nvidia.com/cuda/archive/10.0/pdf/CUDA_C_Programming_Guide.pdf (cit. on pp. 9–12, 87).

[NVI18b] NVIDIA Corporation. *cuSparse Library*. DU-06709-001_v10.0. 2018. URL: https://docs.nvidia.com/cuda/archive/10.0/pdf/CUSPARSE_Library.pdf (cit. on p. 63).

[NVI18c] NVIDIA Corporation. *NVIDIA Tesla V100 GPU Architecture*. Whitepaper. 2018. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (cit. on p. 10).

[NVI18d] NVIDIA Corporation. *NVRTC - CUDA Runtime Compilation*. DU-07529-001_v10.0. 2018. URL: https://docs.nvidia.com/cuda/archive/10.0/pdf/NVRTC_User_Guide.pdf (cit. on p. 83).

[NVI18e] NVIDIA Corporation. *Parallel Thread Execution ISA*. Application Guide. Version 6.3. 2018. URL: https://docs.nvidia.com/cuda/archive/10.0/pdf/ptx_isa_6.3.pdf (cit. on p. 14).

[NVI19] NVIDIA Corporation. *CUDA LLVM Compiler*. 2019. URL: https://developer.nvidia.com/cuda-llvm-compiler (visited on 2019-01-09) (cit. on pp. 78, 84).

[OB11] O'Neil, M. A. and M. Burtscher. "Floating-point Data Compression at 75 Gb/s on a GPU." In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-4. ACM, 2011, 7:1–7:7. DOI: 10.1145/1964179.1964189 (cit. on pp. 78, 99).

[OSV11] Oberhuber, T., A. Suzuki, and J. Vacata. "New Row-grouped CSR Format for Storing Sparse Matrices on GPU with Implementation in CUDA." In: *Acta Technica* 56 (2011), pp. 447–466 (cit. on p. 44).

[OKV15] Odaker, T., D. Kranzlmueller, and J. Volkert. "GPU-Accelerated Real-Time Mesh Simplification Using Parallel Half Edge Collapses." In: *Revised Selected Papers of the 10th International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science. MEMICS 2015*. Vol. 9548. Lecture Notes in Computer Science. 2015, pp. 107–118. DOI: 10.1007/978-3-319-29817-7_10 (cit. on p. 23).

[Omn18] OmniSci. *MapD Rebrands to OmniSci*. CISION PR Newswire. Sept. 27, 2018. URL: https://www.prnewswire.com/news-releases/mapd-rebrands-to-omnisci-300720094.html (visited on 2019-02-18) (cit. on p. 78).

[Ope18a] OpenACC-Standard.org. *The OpenACC® Application Programming Interface*. Version 2.7. 2018. URL: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf (cit. on p. 12).

[Ope18b] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. Version 5.0. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (cit. on p. 12).

[OMGF18] Orme, M., I. Madera, M. Gschweitl, and M. Ferrari. "Topology Optimization for Additive Manufacturing as an Enabler for Light Weight Flight Hardware." In: *MDPI Designs* 2(4) (2018), 51:1–51:22. DOI: 10.3390/designs2040051 (cit. on p. 1).

[PP15] Papageorgiou, A. and N. Platis. "Triangular mesh simplification on the GPU." In: *The Visual Computer* 31(2) (2015), pp. 235–244. DOI: 10.1007/s00371-014-1039-x (cit. on p. 23).

[Per17] Perkins, H. "CUDA-on-CL." In: *Proceedings of the 5th International Workshop on OpenCL*. IWOCL '17. ACM Press, 2017. DOI: 10.1145/3078155.3078156 (cit. on p. 12).

[Pey03] Peyton Jones, S., ed. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. ISBN: 978-0-521-82614-3 (cit. on p. 78).

[RBA+13] Ragan-Kelley, J., C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines." In: *ACM SIGPLAN Notices* 48(6) (2013), pp. 519–530. DOI: 10.1145/2499370.2462176 (cit. on p. 64).

[RKB06] Ratanaworabhan, P., J. Ke, and M. Burtscher. "Fast lossless compression of scientific floating-point data." In: *Proceedings of the Data Compression Conference*. DCC '06. 2006, pp. 133–142. DOI: 10.1109/DCC.2006.35 (cit. on p. 79).

[RBGH14] Rautek, P., S. Bruckner, M. Gröller, and M. Hadwiger. "ViSlang: A System for Interpreted Domain-Specific Languages for Scientific Visualization." In: *Visualization and Computer Graphics, IEEE Transactions on* 20(12) (2014), pp. 2388–2396. DOI: 10.1109/TVCG.2014.2346318 (cit. on p. 78).

[RG15] Reguly, I. Z. and M. B. Giles. "Finite Element Algorithms and Data Structures on Graphical Processing Units." In: *International Journal of Parallel Programming* 43(2) (2015), pp. 203–239. DOI: 10.1007/s10766-013-0301-6 (cit. on p. 45).

[Ren12] Rennich, S. “Leveraging Matrix Block Structure In Sparse Matrix-Vector Multiplication.” In: *NVIDIA GPU Technology Conference*. 2012. URL: http://on-demand.gputechconf.com/gtc/2012/presentations/S0029-Leveraging-Matrix-Block-Structure-Sparse-Matrix-Vector-Multiplication.pdf (cit. on p. 44).

[RB85] Rice, J. R. and R. F. Boisvert. *Solving Elliptic Problems Using ELLPACK*. Springer, 1985. ISBN: 0-387-90910-9 (cit. on pp. 16, 66).

[RSWH98] Richardson, T., Q. Stafford-Fraser, K. R. Wood, and A. Hopper. “Virtual network computing.” In: *IEEE Internet Computing* 2(1) (1998), pp. 33–38. DOI: 10.1109/4236.656066 (cit. on pp. 76, 79).

[Saa03] Saad, Y. *Iterative Methods for Sparse Linear Systems*. 2003. ISBN: 0898715342 (cit. on pp. 15, 64).

[SKR18] Samsel, F., S. Klaassen, and D. H. Rogers. “ColorMoves: Real-time Interactive Colormap Construction for Scientific Visualization.” In: *IEEE Computer Graphics and Applications* 38(1) (2018), pp. 20–29. DOI: 10.1109/mcg.2018.011461525 (cit. on p. 3).

[San16] Sandstorm.io. *Cap’n Proto*. 2016. URL: https://capnproto.org/ (visited on 2016-03-14) (cit. on p. 100).

[SCRL19] Sarton, J., N. Courilleau, Y. Remion, and L. Lucas. “Interactive Visualization and On-Demand Processing of Large Volume Data: A Fully GPU-Based Out-Of-Core Approach.” In: *IEEE Transactions on Visualization and Computer Graphics* (2019). DOI: 10.1109/tvcg.2019.2912752. Early Access (cit. on p. 3).

[SHW04] Schaefer, S., J. Hakenberg, and J. Warren. “Smooth Subdivision of Tetrahedral Meshes.” In: *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry processing*. SGP ’04. 2004, pp. 147–154. DOI: 10.1145/1057432.1057452 (cit. on p. 24).

[SB16] Schmidt, R. and T. Brochu. *Adaptive Mesh Booleans*. May 5, 2016. arXiv: 1605.01760v1 [cs.GR] (cit. on p. 4).

[SA18] Segal, M. and K. Akeley. *The OpenGL® Graphics System: A Specification*. Version 4.6 (Core Profile). 2018. URL: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf (cit. on pp. 11, 86).

[STD+19] Shen, H., F. Tian, X. Deng, C. Xu, A. Rodriguez, I. K., and W. Li. *Intel® CPU Outperforms NVIDIA GPU on ResNet-50 Deep Learning Inference*. May 13, 2019. URL: https://software.intel.com/en-us/articles/intel-cpu-outperforms-nvidia-gpu-on-resnet-50-deep-learning-inference (visited on 2019-09-19) (cit. on p. 97).

[SKSD14] Shen, J., J. Kosinka, M. A. Sabin, and N. A. Dodgson. “Conversion of trimmed NURBS surfaces to Catmull–Clark subdivision surfaces.” In: *Computer Aided Geometric Design* 31(7–8) (2014), pp. 486–498. DOI: 10.1016/j.cagd.2014.06.004 (cit. on p. 23).

[Sho85] Shoemake, K. “Animating Rotation with Quaternion Curves.” In: *SIGGRAPH Computer Graphics* 19(3) (1985), pp. 245–254. DOI: 10.1145/325165.325242 (cit. on p. 62).

[Si15] Si, H. “TetGen, a Delaunay-based Quality Tetrahedral Mesh Generator.” In: *ACM Transactions on Mathematical Software* 41(2) (2015), 11:1–11:36. DOI: 10.1145/2629697 (cit. on p. 70).

[Sie18] Siemens. *NX*. 2018. URL: https://www.plm.automation.siemens.com/global/en/products/nx/ (cit. on p. 70).

[Sod15] Sodani, A. “Knights Landing: 2nd Generation Intel “Xeon Phi” Processor.” In: *Hot Chips: A Symposium on High Performance Chips*. HC27. 2015 (cit. on p. 9).

[Sta98] Stam, J. “Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values.” In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. SIGGRAPH ’98. 1998. DOI: 10.1145/280814.280945 (cit. on p. 24).

[Sta14] Stanford University. *The Stanford 3D Scanning Repository*. 2014. URL: http://graphics.stanford.edu/data/3Dscanrep (cit. on p. 68).

[SDZS16] Steinberger, M., A. Derlery, R. Zayer, and H.-P. Seidel. “How naive is naive SpMV on the GPU?” In: *2016 IEEE High Performance Extreme Computing Conference*. HEPC ’16. 2016. DOI: 10.1109/hpec.2016.7761634 (cit. on p. 67).

[SDS+19] Strohmaier, E., J. Dongarra, H. Simon, M. Meuer, and H. Meuer. *Top500 List*. June 2019. URL: https://www.top500.org/list/2019/06/ (cit. on p. 1).

[Str19] Ströter, D. “Tetrahedral Mesh Processing and Data Structures for Adaptive Volumetric Mesh Booleans on GPUs.” MSc Thesis. Technische Universität Darmstadt, 2019 (cit. on p. 99).

[TDCC17] Tao, D., S. Di, Z. Chen, and F. Cappello. “Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization.” In: *2017 IEEE International Parallel and Distributed Processing Symposium*. IPDPS ’17. 2017. DOI: 10.1109/ipdps.2017.115 (cit. on pp. 78, 100).

[Tas01] Tasora, A. “An Optimized Lagrangian-multiplier Approach for Interactive Multibody Simulation in Kinematic and Dynamical Digital Prototyping.” In: *International Symposium on Computer Simulation in Biomechanics*. VIII ISCSB. 2001 (cit. on pp. 62, 64).

[TPD15] Thébault, L., E. Petit, and Q. Dinh. “Scalable and Efficient Implementation of 3D Unstructured Meshes Computation: A Case Study on Matrix Assembly.” In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP 2015. 2015, pp. 120–129. DOI: 10.1145/2688500.2688517 (cit. on p. 45).

[Tho06] Thompson, L. L. “A Review of Finite-Element Methods for Time-Harmonic Acoustics.” In: *Journal of the Acoustical Society of America* 119(3) (2006), pp. 1315–1330. DOI: 10.1121/1.2164987 (cit. on p. 63).

[TAS06] Tolia, N., D. Andersen, and M. Satyanarayanan. “Quantifying interactive user experience on thin clients.” In: *Computer* 39(3) (2006), pp. 46–52. DOI: 10.1109/MC.2006.101 (cit. on p. 76).

[TRA11] Torsello, A., E. Rodolà, and A. Albarelli. “Multiview Registration via Graph Diffusion of Dual Quaternions.” In: *CVPR 2011*. 2011, pp. 2441–2448. DOI: 10.1109/CVPR.2011.5995565 (cit. on p. 64).

[TS01] Trottenberg, U. and A. Schuller. *Multigrid*. Academic Press, Inc., 2001. ISBN: 0-12-701070-X (cit. on p. 82).

[UMC+19] Urick, B., B. Marussig, E. Cohen, R. H. Crawford, T. J. Hughes, and R. F. Riesenfeld. “Watertight Boolean operations: A framework for creating CAD-compatible gap-free editable solid models.” In: *Computer-Aided Design* 115 (2019), pp. 147–160. DOI: 10.1016/j.cad.2019.05.034 (cit. on p. 2).

[VPM14] Vacondio, R., A. D. Palù, and P. Mignosa. “GPU-enhanced Finite Volume Shallow Water solver for fast flood simulations.” In: *Environmental Modelling & Software* 57 (2014), pp. 60–75. DOI: 10.1016/j.envsoft.2014.02.003 (cit. on p. 87).

[VOFG10] Vázquez, F., G. Ortega, J. J. Fernández, and E. M. Garzón. “Improving the Performance of the Sparse Matrix Vector Product with GPUs.” In: *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*. CIT ’10. 2010, pp. 1146–1151. DOI: 10.1109/CIT.2010.208 (cit. on pp. 44, 51, 63, 66).

[VHS15] Venkat, A., M. Hall, and M. Strout. “Loop and data transformations for sparse matrix code.” In: *ACM SIGPLAN Notices* 50(6) (2015), pp. 521–532. DOI: 10.1145/2813885.2738003 (cit. on p. 65).

[Web19] WebAssembly Community. *WebAssembly Specification*. Version 1.0. 2019. URL: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf (cit. on p. 100).

[Web16] Weber, D. “Interactive Physically Based Simulation — Efficient Higher-Order Elements, Multigrid Approaches and Massively Parallel Data Structures.” PhD thesis. Technische Universität Darmstadt, 2016 (cit. on p. 5).

[WBS+13] Weber, D., J. Bender, M. Schnoes, A. Stork, and D. W. Fellner. “Efficient GPU Data Structures and Methods to Solve Sparse Linear Systems in Dynamics Applications.” In: *Computer Graphics Forum* 32(1) (2013), pp. 16–26. DOI: 10.1111/j.1467-8659.2012.03227.x (cit. on pp. 6, 12, 42, 44, 45, 50, 57, 64, 98).

[WKS+11] Weber, D., T. Kalbe, A. Stork, D. W. Fellner, and M. Goesele. "Interactive Deformable Models with Quadratic Bases in Bernstein-Bézier-form." In: *The Visual Computer* 27(6) (2011), pp. 473–483. DOI: 10.1007/s00371-011-0579-6 (cit. on pp. 43, 46).

[WMA+15] Weber, D., J. S. Mueller-Roemer, C. Altenhofen, A. Stork, and D. W. Fellner. "Deformation Simulation using Cubic Finite Elements and Efficient *p*-Multigrid Methods." In: *Computers & Graphics* 53(Part B) (2015), pp. 185–195. DOI: 10.1016/j.cag.2015.06.010 (cit. on pp. 24, 42, 46, 47, 53).

[WMSF15] Weber, D., J. S. Mueller-Roemer, A. Stork, and D. W. Fellner. "A Cut-Cell Geometric Multigrid Poisson Solver for Fluid Simulation." In: *Computer Graphics Forum* 34(2) (Eurographics 2015), pp. 481–491. DOI: 10.1111/cgf.12577 (cit. on pp. 5, 75, 80, 82).

[WG14] Weber, N. and M. Goesele. "Auto-Tuning Complex Array Layouts for GPUs." In: *Eurographics Symposium on Parallel Graphics and Visualization*. EGPGV '14. 2014. DOI: 10.2312/pgv.20141085 (cit. on p. 65).

[WG17] Weber, N. and M. Goesele. "MATOG: Array Layout Auto-Tuning for CUDA." In: *ACM Transactions on Architecture and Code Optimization* 14(3) (2017), 28:1–28:26. DOI: 10.1145/3106341 (cit. on pp. 62, 65).

[WDV+12] Willberg, C., S. Duczek, J. M. Vivar Perez, D. Schmicker, and U. Gabbert. "Comparison of Different Higher Order Finite Element Schemes for the Simulation of Lamb Waves." In: *Computer Methods in Applied Mechanics and Engineering* 241–244 (2012), pp. 246–261. DOI: 10.1016/j.cma.2012.06.011 (cit. on p. 46).

[Yan19] Yang, C. *GraphBLAST*. 2019. URL: https://github.com/gunrock/graphblast (visited on 2019-02-02) (cit. on p. 29).

[Yax15] Yaxun, L. *[RFC] Proposal for Adding SPIRV Target*. 2015. URL: http://lists.llvm.org/pipermail/llvm-dev/2015-June/086848.html (visited on 2016-03-14) (cit. on p. 84).

[YDT+18] Ye, Z., O. Diamanti, C. Tang, L. Guibas, and T. Hoffmann. "A Unified Discrete Framework for Intrinsic and Extrinsic Dirac Operators for Geometry Processing." In: *Computer Graphics Forum* 37(5) (Symposium on Geometry Processing 2018), pp. 93–106. DOI: 10.1111/cgf.13494 (cit. on p. 63).

[Zak11] Zakai, A. "Emscripten." In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. OOPSLA '11. 2011. DOI: 10.1145/2048147.2048224 (cit. on p. 100).

[ZSS17a] Zayer, R., M. Steinberger, and H.-P. Seidel. "A GPU-Adapted Structure for Unstructured Grids." In: *Computer Graphics Forum* 36(2) (Eurographics 2017), pp. 495–507. DOI: 10.1111/cgf.13144 (cit. on pp. 3, 23, 33, 45, 50, 54, 57, 58, 101).

[ZSS17b] Zayer, R., M. Steinberger, and H.-P. Seidel.
"Sparse Matrix Assembly on the GPU Through Multiplication Patterns."
In: *2017 IEEE High Performance Extreme Computing Conference*. 2017, pp. 1–8.
DOI: 10.1109/HPEC.2017.8091057 (cit. on pp. 3, 5, 45, 50, 54, 55, 57, 58).

[ZG18] Zhang, J. and L. Gruenwald. "Regularizing irregularity: bitmap-based and portable sparse matrix multiplication for graph data on GPUs."
In: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*.
GRADES-NDA '18. 2018. DOI: 10.1145/3210259.3210263 (cit. on p. 66).

[ZZL+16] Zhang, P., M. Zalewski, A. Lumsdaine, S. Misurda, and S. McMillan.
"GBTL-CUDA: Graph Algorithms and Primitives for GPUs."
In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*.
IPDPSW '16. 2016. DOI: 10.1109/ipdpsw.2016.185 (cit. on p. 29).

[ZPL+11] Zhang, Y., L. Peng, B. Li, J.-K. Peir, and J. Chen. "Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications."
In: *IEEE International Symposium on Workload Characterization*. IISWC '11. IEEE, 2011, pp. 205–215. DOI: 10.1109/iiswc.2011.6114180 (cit. on p. 9).

[ZGG+14] Zheng, C., S. Gu, T.-X. Gu, B. Yang, and X.-P. Liu.
"BiELL: A Bisection ELLPACK-based Storage Format for Optimizing SpMV on GPUs."
In: *Journal of Parallel and Distributed Computing* 74(7) (2014). Special Issue on Perspectives on Parallel and Distributed Processing, pp. 2639–2647.
DOI: 10.1016/j.jpdc.2014.03.002 (cit. on p. 44).

[ZLQF15] Zhu, B., M. Lee, E. Quigley, and R. Fedkiw. "Codimensional non-Newtonian Fluids."
In: *ACM Transactions on Graphics* 34(4) (2015), 115:1–115:9. DOI: 10.1145/2766981 (cit. on p. 99).

[ZQC+14] Zhu, B., E. Quigley, M. Cong, J. Solomon, and R. Fedkiw.
"Codimensional Surface Tension Flow on Simplicial Complexes."
In: *ACM Transactions on Graphics* 33(4) (2014), 111:1–111:11.
DOI: 10.1145/2601097.2601201 (cit. on p. 99).

[ZT00] Zienkiewicz, O. C. and R. L. Taylor. *Finite Element Method: Volume 1*. Fifth.
Butterworth-Heinemann, 2000. ISBN: 0-750-65049-4 (cit. on pp. 15, 45–47).

[ZWPG03] Zorriassatine, F., C. Wykes, R. Parkin, and N. Gindy.
"A survey of virtual prototyping techniques for mechanical product development."
In: *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture* 217(4) (2003), pp. 513–530. DOI: 10.1243/095440503321628189
(cit. on p. 1).

A. Glossary

ABI Application Binary Interface. 65

ALU Arithmetic Logic Unit. 9

AM Additive Manufacturing. iv, 1, 43, 98

AoS Array of Structures. 64–68, 70, 72, 73

AoSoA Array of Structures of Arrays. 64–67

API Application Programming Interface. 2, 3, 11, 12, 14

Array HFDS Array-based Half-Face Data Structure. 22

B-rep Boundary Representation. 2, 19, 20, 61

BSR Block Compressed Sparse Row. 52, 64, 70, 73

$\mathbb{C}$ The field of complex numbers. 62

CAD Computer-Aided Design. iv, v, 2, 4, 9, 19, 23, 61

CAE Computer-Aided Engineering. 2, 62, 76, 98

CAx Computer-Aided technologies. iv, v, 2, 19, 20

CC Compute Capability. 12–14, 44, 68, 70

CFD Computational Fluid Dynamics. 76, 80, 100

COO Coordinate list format. 15, 16

CPU Central Processing Unit. The system processor. 2–5, 9–11, 14–16, 20, 21, 23, 26, 35, 37, 38, 42, 54, 58, 64, 65, 70, 75, 77, 80, 83, 84, 86–88, 91–95, 97, 100, 101

CSC Compressed Sparse Column. 16, 17, 23, 28, 65, 67, 101

CSM Computational Structural Mechanics. 80, 100

CSR Compressed Sparse Row. 16, 17, 22, 25, 26, 28, 39, 44, 45, 51, 52, 62, 64–68, 70, 74, 98, 101

$\boldsymbol{\partial}_k$ The discrete boundary operator, represented as a ternary sparse matrix, that describes the top down relationships from each k-face to its oriented $(k-1)$-face facets. 24–27, 29, 31–34, 36, 50

$\hat{\boldsymbol{\partial}}_k^n$ The indirect top down relationships, represented as a boolean sparse matrix, from each k-face to the non-oriented $(k-n-1)$-faces that it contains. 25, 27, 29, 31, 33, 34, 36, 50

DCEL Doubly Connected Edge List. Also called HEDS. 21, 22, 124

$\mathbf{d}_k$ The discrete coboundary operator, represented as a ternary sparse matrix, that describes the bottom up relationships from each $(k-1)$-face facet to the k-faces that contain the facet. 25, 27, 29, 35, 36, 50

$\hat{\mathbf{d}}_k^n$ The indirect bottom up relationships, represented as a boolean sparse matrix, from each $(k-n-1)$-face to the non-oriented k-faces that contain it. 25, 27, 29, 32, 35, 49, 50

DDG Discrete Differential Geometry. 24, 25

DEC Discrete Exterior Calculus. 24, 25

dG Discontinuous Galerkin. 47

DRAM Dynamic Random-Access Memory. 11, 13, 15

DSL Domain Specific Language. 64, 78

eDSL Embedded Domain Specific Language. 64, 78

ELL ELLPACK-ITPACK format. 16, 17, 44, 45

$\mathbb{F}$ An arbitrary field such as the field of real numbers $\mathbb{R}$. 15, 16

FCRC Flight Crew Rest Compartment. 41

FEM Finite Element Method. iv, v, 2, 3, 7, 15, 21, 24, 42–46, 50, 58, 62, 64, 70, 95, 96, 99

FLOPS Floating Point Operations per Second. 1, 56

FPGA Field Programmable Gate Array. 1, 11

FVM Finite Volume Method. 5, 7, 15, 75, 81, 82

Gi The ISO/IEC 80000 binary unit prefix "gibi" with a value of 1024^3. 35, 54, 56, 70, 87, 88

GPGPU General Purpose computing on the Graphics Processing Unit. 2–4, 7, 9, 11, 17, 20, 21, 23, 42, 43, 63, 77, 78, 93, 95–97

GPU Graphics Processing Unit. iv, v, 1–7, 9–17, 20, 21, 23–26, 29, 31, 35–39, 42–46, 50, 52, 54, 56–58, 62–66, 68, 70, 74–80, 83, 84, 86–88, 91–100

$\mathbb{H}$ The division ring of quaternions. 62, 65, 66, 70

HEDS Half-Edge Data Structure. Also called DCEL. 22, 123

HPC High Performance Computing. 1–3, 42, 45, 76, 79

HTML5 HyperText Markup Language Version 5. 5, 80, 82, 88, 93

ISA Instruction Set Architecture. 14

JIT Just-In-Time. 2, 14, 63, 64, 80, 92, 97, 100

JSON JavaScript Object Notation. 100

Ki The ISO/IEC 80000 binary unit prefix “kibi” with a value of 1024. 11, 14, 35, 53, 92

KNL Intel Xeon Phi Knight’s Landing. 9

Mi The ISO/IEC 80000 binary unit prefix “mebi” with a value of 1024^2. 35

NURBS Non-Uniform Rational B-Splines. 21, 46

OVM OpenVolumeMesh. 21, 22, 35–39

PCIe Peripheral Component Interface Express. 15, 17, 26, 76, 77, 79, 87, 92–94, 97

PDE Partial Differential Equation. 15

PTX Parallel Thread Execution. 14, 77

$\mathbb{R}$ The field of real numbers. 24, 27, 46, 47, 63, 65, 66, 70, 124

RDP Remote Desktop Protocol. 79

Rixel Rich Pixel. 5, 75, 77, 78, 80, 86–88, 92, 93, 96, 100

SIMD Single Instruction Multiple Data. 9, 10, 97

SIMT Single Instruction Multiple Thread. 10, 97

SM Streaming Multiprocessor. 10, 11, 13, 67, 68, 70, 97

SMT Simultaneous Multithreading. 11, 97

SoA Structure of Arrays. 64–67, 72, 73

SoC System on a Chip. 77, 98

SpGEMM General Sparse Matrix-Matrix product. 15, 25, 28, 44, 45, 54, 82, 98

SPMD Single Program Multiple Data. 12

SpMV Sparse Matrix-Vector product. 11, 15, 16, 44, 52, 58, 63, 67, 68, 70, 73, 74

TCP Transmission Control Protocol. 82, 89

TCSR Ternary Compressed Sparse Row. 6, 24–28, 31, 33–35, 38, 95, 98, 101

UDP User Datagram Protocol. 82

VNC Virtual Network Computing. 76, 79, 89

B. Publications and Talks

B.1. Authored and Co-Authored Publications

[ABH+14] Adachi, S., P. Brandstätt, W. Herget, P. Leistner, V. Landersheim, D. Weber, and **J. S. Mueller-Roemer**.
"Wind tunnel test and CFD/CAA analysis on a scaled model of a nose landing gear."
In: *Greener Aviation. Clean Sky breakthroughs and worldwide status*. Greener Air. 2014.

[WMA+14] Weber, D., **J. S. Mueller-Roemer**, C. Altenhofen, A. Stork, and D. W. Fellner. "A p-Multigrid Algorithm using Cubic Finite Elements for Efficient Deformation Simulation."
In: *Workshop on Virtual Reality Interaction and Physical Simulation*. VRIPHYS '14. 2014, pp. 49–58. DOI: 10.2312/vriphys.20141223.

[WMS+14] Weber, D., **J. S. Mueller-Roemer**, J. Simpson, S. Adachi, W. Herget, V. Landersheim, and D. Laveuve.
"Smart Droop Nose for application to Laminar Wing of future Green Regional A/C."
In: *Greener Aviation. Clean Sky breakthroughs and worldwide status*. Greener Air. 2014.

[WMA+15] Weber, D., **J. S. Mueller-Roemer**, C. Altenhofen, A. Stork, and D. W. Fellner.
"Deformation Simulation using Cubic Finite Elements and Efficient p-Multigrid Methods."
In: *Computers & Graphics* 53(Part B) (2015), pp. 185–195.
DOI: 10.1016/j.cag.2015.06.010.

[WMSF15] Weber, D., **J. S. Mueller-Roemer**[1], A. Stork, and D. W. Fellner.
"A Cut-Cell Geometric Multigrid Poisson Solver for Fluid Simulation."
In: *Computer Graphics Forum* 34(2) (Eurographics 2015), pp. 481–491.
DOI: 10.1111/cgf.12577.

[MA16] **Mueller-Roemer, J. S.** and C. Altenhofen.
"JIT-compilation for Interactive Scientific Visualization."
In: *Short Papers Proceedings: 24th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*. WSCG '16. 2016, pp. 197–206.

[ALM+17] Altenhofen, C., F. Loosmann, **J. S. Mueller-Roemer**, T. Grasser, T. H. Luu, and A. Stork.
"Integrating Interactive Design and Simulation for Mass Customized 3D-Printed Objects – A Cup Holder Example." In: *Proceedings of the 28th Annual International Solid Freeform Fabrication Symposium – An Additive Manufacturing Conference*. SFF '17. 2017, pp. 2289–2301.

[MAS17] **Mueller-Roemer, J. S.**, C. Altenhofen, and A. Stork. "Ternary Sparse Matrix Representation for Volumetric Mesh Subdivision and Processing on GPUs." In: *Computer Graphics Forum* 36(5) (Symposium on Geometry Processing 2017), pp. 59–69.
DOI: 10.1111/cgf.13245.

[1] The two primary authors contributed equally to this work.

[ALG+18] Altenhofen, C., T. H. Luu, T. Grasser, M. Dennstädt, **J. S. Mueller-Roemer**, D. Weber, and A. Stork. "Continuous Property Gradation for Multi-material 3D-printed Objects." In: *Proceedings of the 29th Annual International Solid Freeform Fabrication Symposium – An Additive Manufacturing Conference*. SFF '18. 2018, pp. 1675–1685.

[MS18] **Mueller-Roemer, J. S.** and A. Stork. "GPU-based Polynomial Finite Element Matrix Assembly for Simplex Meshes." In: *Computer Graphics Forum* 37(7) (Pacific Graphics 2018), pp. 443–454. DOI: 10.1111/cgf.13581.

[VLM+18] Volk, R., T. H. Luu, **J. S. Mueller-Roemer**, N. Sevilmis, and F. Schultmann. "Deconstruction Project Planning of Existing Buildings Based on Automated Acquisition and Reconstruction of Building Information." In: *Automation in Construction* 91 (2018), pp. 226–245. DOI: 10.1016/j.autcon.2018.03.017.

[BGM19] Bormann, P., R. Gutbell, and **J. S. Mueller-Roemer**. "Integrating Server-based Simulations into Web-based Geo-applications." In: *Eurographics 2019 - Short Papers*. 2019. DOI: 10.2312/egs.20191012.

[MSF19] **Mueller-Roemer, J. S.**, A. Stork, and D. W. Fellner. "Joint Schedule and Layout Autotuning for Sparse Matrices with Compound Entries on GPUs." In: *Vision, Modeling and Visualization*. VMV '19. 2019, pp. 109–116. DOI: 10.2312/vmv.20191324.

B.2. Talks

1. Eurographics Symposium on Geometry Processing (SGP) 2017: Ternary Sparse Matrix Representation for Volumetric Mesh Subdivision and Processing on GPUs. London, United Kingdom, 2017.
2. Pacific Graphics 2018: GPU-based Polynomial Finite Element Matrix Assembly for Simplex Meshes. Hong Kong, China, 2018.
3. 24th International Symposium on Vision, Modeling and Visualization (VMV) 2019: Joint Schedule and Layout Autotuning for Sparse Matrices with Compound Entries on GPUs. Rostock, Germany, 2019.

C. Supervisory Activities

C.1. Master's Theses

[Luu16] Luu, T. H. "Adaptives und hybrides SLAM für handgeführte RGBD-Kameras." MSc Thesis. Technische Universität Darmstadt, 2016.

[Str19] Ströter, D. "Tetrahedral Mesh Processing and Data Structures for Adaptive Volumetric Mesh Booleans on GPUs." MSc Thesis. Technische Universität Darmstadt, 2019.

C.2. Bachelor's Theses

[Dit19] Ditzinger, A. "GPU-accelerated Heat Simulation using Block-Structured FE Meshes." BSc Thesis. Technische Universität Darmstadt, 2019. Preliminary title of thesis in progress.

[Lot18] Lotter, L. "GPU-basierte Shallow Water Simulation." BSc Thesis. Technische Universität Darmstadt, 2018.

C.3. Additional Supervisory Activities

1. Supervision of several hands-on training projects in the (Advanced) Visual Computing Lab courses (TU Darmstadt TUCaN IDs 20-00-0418-pr and 20-00-0537-pr).
2. Supervision of several hands-on training projects in the Project Lab Programming Massively Parallel Systems (TU Darmstadt TUCaN ID 20-00-0763-pp).
3. Teaching assistant for the Physically-based Simulation and Animation course (TU Darmstadt TUCaN ID 20-00-0682-iv).