Vision, Modeling, and Visualization (2021)
B. Andres, M. Campen, and M. Sedlmair (Eds.)

GPU-Parallel Constant-Time Limit Evaluation of Catmull-Clark Solids

S. Besler[1], C. Altenhofen[1], A. Stork[1] and D. W. Fellner[1,2]

[1]TU Darmstadt & Fraunhofer IGD, Germany
[2]TU Graz, Austria

Abstract

Subdivision solids, such as Catmull-Clark (CC) solids, are versatile volumetric representation schemes that can be employed for geometric modeling, physically based simulation, and multi-material additive manufacturing. With volumetric limit evaluation still being the performance bottleneck for these applications, we present a massively parallel approach to Altenhofen et al.'s constant-time limit evaluation method for CC solids. Our algorithm exploits the computational power of modern GPUs, while maintaining the mathematical concepts of Altenhofen et al.'s method. Distributing the computations for a single cell across multiple streaming multiprocessors (SMs) increases the utilization of the GPU's resources compared to straightforward parallelization. Specialized compute kernels for different topological configurations optimize shared memory usage and memory access. Our hybrid approach dynamically chooses the best kernel based on the topology and the evaluation parameters, resulting in speedups of between $5.75\times$ *and* $61.58\times$ *compared to a CPU-parallel implementation of Altenhofen et al.'s method.*

1. Introduction

While subdivision surfaces are widely used in the entertainment industry, especially in the fields of computer graphics and computer animation, they have also been receiving increased attention in engineering in recent years. Generative design and topology optimization – both boosted by the recent advancements in additive manufacturing (AM) – create a need for flexible geometric representations suited for organic shapes. At the same time, additive manufacturing allows for creating objects with locally varying volumetric properties, such as density or stiffness, complex inner structures, and advanced functionality that cannot be achieved by traditional manufacturing techniques, such as milling or casting.

Subdivision solids – the volumetric extension of the concepts behind subdivision surfaces – provide flexible volumetric representation schemes that allow for design, analysis, and manufacturing such objects. Recent publications show how subdivision solids, i.e. Catmull-Clark (CC) solids [JM99] can be employed for geometric modeling [ASSF17], physically based simulation via isogeometric analysis (IGA) [BHU10, XXD+20, AESF21], and multi-material additive manufacturing [ALG+18, LAE+19].

The main challenge when working with subdivision solids is efficiently evaluating the limit volume, especially in irregular regions. For Catmull-Clark solids, Altenhofen et al. presented a limit evaluation approach [AMW+18] that, similar to Stam's approach for CC surfaces [Sta98a], evaluates every individual limit point in constant time. Nevertheless, evaluating the limit volume is still the computational bottleneck when using CC solids for all three fields mentioned above – geometric modeling, IGA, multi-material AM. Decreasing the simulation times for CC-solid IGA would allow for analyzing more design variants in the same time in an optimization scenario and would enable interactive modeling environments to provide immediate feedback on the design and especially on design changes based on simulation results. Recent developments in additive manufacturing, i.e. *digital light processing* (DLP) and *direct image sintering* (DIS) allow for processing an entire layer of material at once instead of curing or sintering the material within the layer along a given path. As this has tremendous potential in reducing printing times, the slicing process has to be sped up accordingly to not become a bottleneck.

Research has oftentimes succeeded in accelerating compute intensive tasks by parallelization and by utilizing massively parallel processors such as graphics processing units (GPUs). GPUs can perform generic computing tasks in parallel using a GPGPU computing framework, such as NVIDIA CUDA [NVI21a] or OpenCL [NVI21b]. As the algorithm by Altenhofen et al. [AMW+18] shows little interdependency between the limit of different cells in the CC-solid model, as well as between the limit points of a single cell, the algorithm can benefit greatly from massively parallel computation exploiting the computational power of GPUs. As limit evaluation can take a significant part of the computation time in the aforementioned applications, massively parallel computation of the limit volume will greatly benefit them.

Our main contributions are:

- A massively parallel constant-time limit evaluation approach for

DOI: 10.2312/vmv.20211369

Catmull-Clark solids, exploiting the computational throughput of modern GPUs
- An approach to distribute the evaluation of a layered or irregular cell across multiple streaming multiprocessors (SMs), ensuring a sufficient workload for the GPU, even when evaluating just a few cells
- Specialized compute kernels optimizing shared memory usage and computation time for different topological configurations and evaluation parameters
- A *hybrid* strategy that dynamically chooses the optimal kernel for minimizing computation time by exploiting the advantages of each kernel type

2. Related Work

The basic concept behind subdivision algorithms is to iteratively refine a given control mesh following a set of *subdivision rules*. The specific rules depend on the subdivision scheme to be used. For most schemes, this refinement process converges to the so-called *limit* for an infinite number of subdivision steps. Various methods for curves, surfaces and solids have been presented over the years, each with different requirements regarding the topology of the control mesh and the properties, e.g. smoothness, of the resulting limit surface. While e.g. the Loop subdivision scheme [Loo87] requires purely triangular meshes, Catmull-Clark subdivision [CC78] can handle more complex surface meshes with arbitrary polygons.

Since performing an infinite number of subdivision steps cannot be done in real world applications, the limit surface can only be approximated by explicit subdivision. However, e.g. for Loop and Catmull-Clark surfaces, Stam's groundbreaking methods [Sta98b, Sta98a] allow for efficiently evaluating the limit surface directly from the control mesh. His algorithms are one of the reasons for the wide use of subdivision surfaces in the entertainment industry.

While subdivision surfaces only describe the shape of a 3D object by means of its outer surface, subdivision solids provide a volumetric representation with cells and inner control points for internal degrees of freedom and encoding volumetric information. Following the same concepts as subdivision surfaces, subdivision solids feature a set of subdivision rules that converge towards the so-called *limit volume*. Different polyhedral meshes can serve as control meshes for different volumetric subdivision schemes, such as Schaefer et al.'s scheme [SHW04] on tetrahedra, or Bajaj et al.'s scheme [BSWX02] for hexahedral meshes. Catmull-Clark solids as presented by Joy and MacCracken [JM99] support control meshes with almost arbitrary polyhedra (see Section 3.1). An overview of the individual subdivision schemes, their benefits and drawbacks can be found in Chang's and Qin's survey paper [CQ03].

Building on Stam's efficient limit evaluation approach [Sta98a] and incorporating Liu et al.'s insights on irregular topological configurations [LZC+18], Altenhofen et al. [AMW+18] presented an efficient limit evaluation method for Catmull-Clark solids. Similar to Stam's approach for CC surfaces, it evaluates every limit point in constant time regardless of its proximity to extraordinary vertices or edges, resulting in a linear complexity w.r.t. the total number of limit points to be evaluated. However, volumetric limit evaluation is still the performance bottleneck. Even though Luu et al. use the constant-time limit evaluation, the performance of their algorithm for slicing multi-material CC-solid models [LAE+19] is still limited by the evaluation process.

While researchers such as Hughes et al., Dokken et al. and Elber et al. worked on employing trivariate tensor-product splines, e.g. B-splines and NURBS volumes to geometric modeling [ME16], IGA [HCB05, DSHB09], and multi-material AM [EDE17], these representation schemes are out of scope for our paper. However, especially for IGA, using tensor-product splines is much more common than using subdivision models.

To the best of our knowledge, no massively parallel algorithms exist for evaluating the limit volume of subdivision solids. However, researchers have worked on accelerating the evaluation of subdivision surfaces. Bolz et al. [BS02] presented a parallelized technique for Catmull-Clark surfaces, using lookup tables to rapidly evaluate the limit. Nießner et al. [NLMD12] use a combination of compute and tessellation shaders on the GPU to efficiently evaluate the limit surface during rendering. For subdivision solids, Mueller-Roemer et al. [MAS17] showed how to efficiently store and process volumetric meshes on the GPU, using ternary sparse matrices to represent the CC-solid control mesh and presenting a massively parallel implementation of Catmull-Clark solid subdivision. Volumetric limit evaluation has not yet been performed on GPUs.

3. Catmull-Clark Solids

As our approach performs massively parallel limit evaluation on Catmull-Clark solids, we briefly introduce the concepts of CC solids and of Altenhofen et al.'s constant-time limit evaluation technique [AMW+18].

3.1. Terms and Concepts

Volumetric control meshes for CC solids are polyhedral meshes defined by vertices, edges, faces, and cells. One of the advantages of Catmull-Clark solid subdivision compared to the volumetric subdivision schemes of e.g. Schaefer et al. [SHW04] and Bajaj et al. [BSWX02] is the flexibility of the topology of the control mesh. CC-solid control meshes can consist of any type of polyhedron that subdivides purely into hexahedra after at most two subdivision steps. Such polyhedra are e.g. tetrahedra, wedges, extruded polygons with quadrilateral side faces, and more complex polyhedra as shown by Liu et al. [LZC+18]. n-sided pyramids with $n >= 4$ do not subdivide into hexahedra and are therefore not supported in CC-solid control meshes.

When evaluating the limit volume of a CC-solid model, regions in the control mesh that deviate from regular hex-mesh topology require additional attention. While Catmull-Clark surfaces define *extraordinary vertices* (EVs) as control points with a valence other than 4 – the valence being the number of edges attached to the vertex – CC solids transfer that definition to the volumetric case and also introduce *extraordinary edges* (EEs). An EE is an edge with a valence other than 4 for internal edges and a valence other than 3 for edges at the boundary of the control mesh. The valence of an edge is defined by the number of attached faces. Any control point, where multiple EEs meet for a single cell, is defined as an

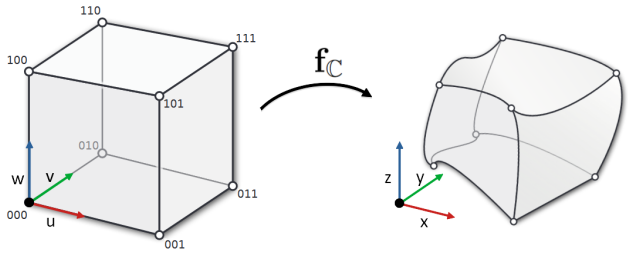
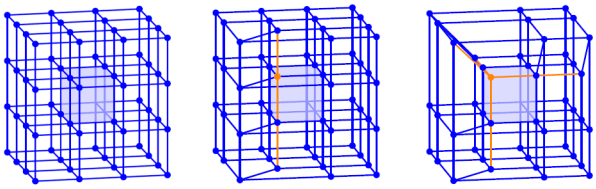


Figure 1: *The transformation* $\mathbf{f}_{\mathbb{C}}$ *of each CC-solid cell* $\mathbb{C}$ *from parameter space* $(u,v,w) \in [0,1]^3$ *to its limit volume in Euclidean space* $\mathbb{R}^3$. *Image adopted from Altenhofen et al. [AESF21].*



(a) Regular cell (b) Layered cell (c) Irregular cell

Figure 2: *Local control meshes for the three cell classes defined by Altenhofen et al. [AMW+18], with EEs and EVs marked in orange. The cell to be evaluated is visualized in light blue. In the examples shown here, all EEs have a valence of* 3. *Images adopted from Altenhofen et al. [AMW+18].*

EV. For the CC-solid limit volume to be evaluatable, each cell of the control mesh must be hexahedral and may contain at most one EV. If a cell contains multiple EEs, they must all share a common EV.

Each CC-solid cell $\mathbb{C}$ is parametrized as a unit cube in parameter space $(u,v,w) \in [0,1]^3$. The CC-solid basis functions describe the transformation $\mathbf{f}_{\mathbb{C}}$ that maps every parameter point (u,v,w) to its limit position (x,y,z) in Euclidean space $\mathbb{R}^3$ (see Figure 1).

3.2. Volumetric Limit Evaluation

The goal of limit evaluation of Catmull-Clark solids is to accurately compute the limit volume of a CC-solid model without explicitly subdividing the control mesh. The limit volume is evaluated per cell. For each cell, the limit is computed at a set of sample points (u,v,w), each sample point resulting in exactly one limit point (x,y,z). The limit point is thereby defined by the local control points and the subdivision basis functions evaluated at its parameters (u,v,w). Based on their topological configuration, the cells of a CC-solid model are categorized into three different classes that define their subdivision basis functions in different ways.

Regular cells are cells that do not contain any extraordinary edges. For cells inside the CC-solid model, the local control points form a regular grid of $4 \times 4 \times 4$ points as shown in Figure 2(a), leading to 64 control points. For regular boundary cells, the topmost layer does not exist, resulting in a grid of $3 \times 4 \times 4$ control points. The limit points $p_{regular}(u,v,w)$ of a regular cell are calculated as the product of its control points $\mathbf{C}$ and a tensor product of three univariate B-Spline basis functions $\mathbf{N}(u,v,w)$:

$$p_{regular}(u,v,w) = \mathbf{C}^T \mathbf{N}(u,v,w) \tag{1}$$

Layered cells have one extraordinary edge and form a layered local control mesh as shown in Figure 2(b). Each layer represents an irregular 2D control mesh with the same valence as the EE. Therefore, the limit evaluation uses the 2D eigenstructures known from Stam's Catmull-Clark surface evaluation [Sta98a] to compute the basis functions. The limit points $p_{layered}(u,v,w)$ of a layered cell are given by the product of the local control points $\mathbf{C}$ and the tensor product of Stam's 2D subdivision basis functions $\varphi(u,v)$ with an univariate cubic B-spline basis function $\mathbf{N}(w)$:

$$p_{layered}(u,v,w) = \mathbf{C}^T \varphi(u,v) \mathbf{N}(w) \tag{2}$$

Irregular cells have more than one EE and therefore also an EV. The topology of their local control mesh depends on the combination of valences of the EEs. Figure 2(c) shows an irregular cell with two EEs, each with a valence of 3. Liu et al. presented a method for algorithmically enumerating all possible irregular combinations, showing examples for the valences 3 and 5 [LZC+18]. Evaluating the limit of such cells requires the calculation of the local volumetric subdivision matrix and its 3D eigenstructure as described by Altenhofen et al. [AMW+18]. With each local subdivision step, the cell is split into a minimum of four regular sub-cells and a maximum of three layered sub-cells. The sub-cell that contains the extraordinary vertex remains irregular and features the same topological configuration as the original cell. Evaluating a limit point in an irregular cell therefore results in locally subdividing the cell until the parameter point (u,v,w) lies in a regular or layered sub-cell. The limit point can then be evaluated using the aforementioned strategies. Similar to Stam's approach, all local subdivision steps are combined into a single mathematical operation, using a consistent local subdivision matrix and its 3D eigenstructure. Finally, the limit points are computed via the product of the control points $\mathbf{C}$, the 3D eigenstructure and the volumetric basis functions $\mathbf{N}$ of the corresponding regular or layered sub-cell:

$$p_{irregular}(u,v,w) = \mathbf{C}^T \mathbf{V}^{-T} \Lambda^{n-1} \mathbf{V}^T \bar{\mathbf{A}}^T \mathbf{P}_k^T \mathbf{N}(\phi_{k,n}(u,v,w)) \tag{3}$$

Directly evaluating regular and layered cells as well as using the 3D eigenstructure for evaluating irregular cells results in a constant-time volumetric limit evaluation for CC-solid models. The approach works for both interior and boundary cells. Furthermore, it allows for evaluating the derivatives of the basis functions required for IGA and multi-material AM slicing. The derived basis functions at any given parameter point (u,v,w) are computed by substituting the B-spline basis functions with their partial derivatives. More details on volumetric limit evaluation can be found in the paper by Altenhofen et al. [AMW+18].

4. Massively Parallel Limit Evaluation

Limit evaluation is a computationally expensive process heavily relying on floating point arithmetic. Since current generation GPUs

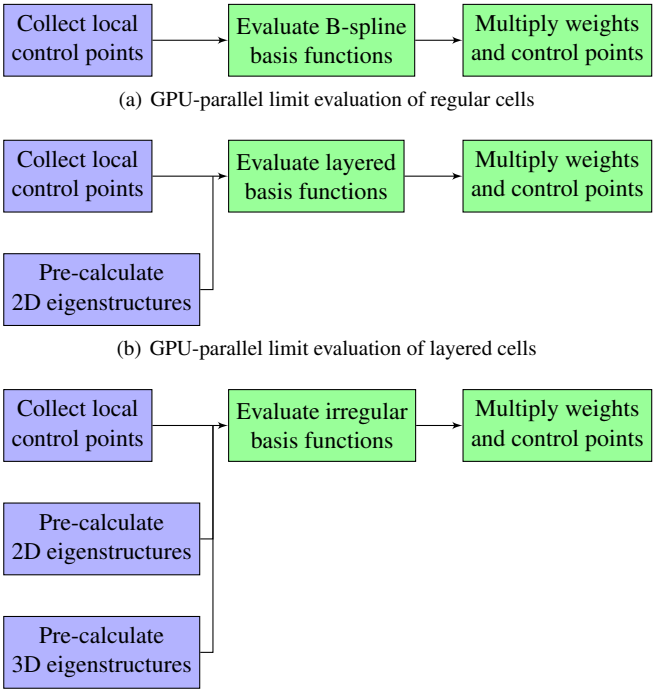
feature higher arithmetic throughput than CPUs, a massively parallel implementation exploiting the computational power of GPUs is expected to drastically reduce the runtime of the limit evaluation algorithm. However, massively parallel computation requires independent computations and data subsets to make full use of the high arithmetic throughput of the massively parallel processor.

In our massively parallel limit evaluation approach, we designed separate evaluation algorithms for each cell class to allow fine-grained optimization of our methods to the requirements of each class.

4.1. Pipeline

Our method uses the CPU to categorize the cells and precompute the needed 2D and 3D eigenstructures. The computation of basis functions and limit points is executed on the GPU in parallel using the NVIDIA CUDA toolkit [NVI21a].

Categorizing the cells into the three classes – regular, layered and irregular – requires analyzing the topology of the volumetric model, i.e. the one-ring neighborhood of every cell. We do this by traversing a pointer-based half-face data structure that describes the CC-solid object. To calculate the 2D and 3D eigenstructures, the corresponding local subdivision matrices are constructed and their eigenvalues and eigenvectors are computed. During this step, our algorithm divides the 2D eigenstructures needed for evaluating layered cells and 3D eigenstructures of irregular cells into submatrices as described in Section 4.2.2. The subdivision basis functions and limit points are finally computed on the GPU. Figure 3 shows the evaluation pipelines for the three cell classes.



(a) GPU-parallel limit evaluation of regular cells

(b) GPU-parallel limit evaluation of layered cells

(c) GPU-parallel limit evaluation of irregular cells

Figure 3: *Pipelines of our GPU-parallel limit evaluation technique for each cell class – regular, layered and irregular. Steps that are executed on the CPU are colored in lavender, while steps that are performed on the GPU are shown in green.*

4.2. Parallelization

Following the NVIDIA CUDA programming model for modern NVIDIA GPUs [NVI20a], work is organized in a *grid* of *blocks*, containing groups of *threads*, all of which execute the same program – the so-called *kernel*. The blocks of a grid are scheduled onto the *streaming multiprocessors (SMs)* – hardware units on the GPU, that contain multiple cores and on-chip shared memory, accessible from all cores of the SM. The threads of each block are executed by the cores of an SM. As SMs do not feature a scheduler per core, threads are grouped into clock-synchronous execution units of 32 threads, called *warps*.

The programming model defines the hierarchy of threads and defines data exchange between threads. The threads of a block can use the *shared memory* of the SM to store and exchange data. Data exchange between the blocks of a grid or between the GPU and CPU is done via *global memory* – the GPUs DRAM. Threads belonging to the same warp can also make use of *warp-shuffle* operations to exchange data stored in registers.

Our GPU-parallel limit evaluation uses one block per regular cell. For layered and irregular cells, multiple blocks are used to compute the limit of each cell.

The actual computation of the limit points is parallelized within each block in two different ways, again based on the cell's topology. The *thread-based* approach computes one limit point per thread. This approach maximizes the multiply-add instruction to memory instruction ratio by maximizing the work per thread. The *warp-based* approach allocates an entire warp to compute a limit point, maximizing parallelism, while also optimizing global memory access across the warp. As we provide individual kernels per cell class, our method features a *hybrid* approach that dynamically chooses between the thread-based and the warp-based kernel for each cell class, based on the number of cells in the corresponding cell class and the number of sample points per cell.

As the block size cannot exceed a hardware limit of 1024 threads [NVI20a] and the performance of a compute kernel is impacted by the block size, the implementation limits the block size to reach optimal performance. Our experiments showed that the performance of the kernels peaks at a maximum block size of 128. If the number of limit points per cell exceeds the maximum thread count, the threads of each block compute the limit points using a block-stride loop.

4.2.1. Regular Cells

The massively parallel limit evaluation uses a single block to compute all limit points of a regular cell. Each thread – or warp, respectively – computes one or more limit points of the cell by evaluating Equation (1).

Figure 4 shows the calculations for evaluating a limit point p

$$p = \mathbf{C}_0^T \mathbf{N} = \begin{pmatrix} \mathbf{c}_1 & \dots & \mathbf{c}_{32} & \mathbf{c}_{33} & \dots & \mathbf{c}_{64} \end{pmatrix} \begin{pmatrix} n_1 \\ \vdots \\ n_{32} \\ n_{33} \\ \vdots \\ n_{64} \end{pmatrix}$$

$$p_{thread} = \overbrace{\begin{pmatrix} \mathbf{c}_1 n_1 + \ldots + \mathbf{c}_{32} n_{32} + \mathbf{c}_{33} n_{33} + \ldots + \mathbf{c}_{64} n_{64} \end{pmatrix}}^{\text{thread}_1}$$

$$p_{warp} = \overbrace{\begin{pmatrix} \mathbf{c}_1 & \mathbf{c}_{33} \end{pmatrix} \begin{pmatrix} n_1 \\ n_{33} \end{pmatrix}}^{\text{thread}_1} + \ldots + \overbrace{\begin{pmatrix} \mathbf{c}_{32} & \mathbf{c}_{64} \end{pmatrix} \begin{pmatrix} n_{32} \\ n_{64} \end{pmatrix}}^{\text{thread}_{32}}$$

$$p_{warp} = \begin{pmatrix} \mathbf{c}_1 n_1 + \mathbf{c}_{33} n_{33} + \ldots + \mathbf{c}_{32} n_{32} + \mathbf{c}_{64} n_{64} \end{pmatrix}$$

Figure 4: *Limit evaluation of a regular cell using the thread-based as well as the warp-based evaluation approach. The thread-based approach computes one limit point in a single thread, while the warp-based approach uses an entire warp to compute the limit point.*

inside a regular cell using the thread-based and the warp-based approaches. The control points as well as the number of basis functions along each parameter axis u, v, w are stored in shared memory, as they are constant across all limit points of the cell. The warp-based kernel also stores the basis functions along each parameter axis in shared memory, as these are accessed by all threads of the warp. As the number of control points and basis functions of regular cells will either be 64 for inner cells or 48 for boundary cells, we optimize shared memory consumption of the compute kernels by allocating shared memory based on the local topology (internal or boundary). Our implementation features a thread-based kernel and a warp-based kernel for internal cells (*REG*64), as well as a thread-based kernel and a warp-based kernel for boundary cells (*REG*48).

4.2.2. Eigenstructure Tiling

In contrast to the evaluation of regular cells, the evaluation of layered and irregular cells uses 2D and 3D eigenstructures as described in Section 3.2. To reduce access times and increase throughput of the computation, the 2D and 3D eigenstructures of layered and irregular cells, are stored in shared memory. Due to the arbitrary size of the eigenstructures – depending on the valences of the EEs – and the limited amount of shared memory available on each SM, a single eigenstructure can limit parallelism by consuming the entire shared memory of a block or can even exceed the amount of shared memory available per block. Therefore, our method uses tiles – sub-matrices with fixed sizes – of the eigenstructures to compute the limit points of layered and irregular cells. Assigning each tile to a different block ensures that the eigenstructures can be kept in shared memory, regardless of their dimensions, while also maintaining parallelism by distributing the eigenstructures across multiple blocks. Figures 5(a) and 5(b) show the tiling of 2D and

$$\mathbf{E} = \begin{pmatrix} e_{1,1} & \cdots & e_{1,16} \\ \vdots & \ddots & \vdots \\ e_{32,1} & \cdots & e_{32,16} \\ e_{33,1} & \cdots & e_{33,16} \\ \vdots & \ddots & \vdots \\ e_{64,1} & \cdots & e_{64,16} \end{pmatrix} \quad \mathbf{E}_{1,1} = \begin{pmatrix} e_{1,1} & \cdots & e_{1,16} \\ \vdots & \ddots & \vdots \\ e_{32,1} & \cdots & e_{32,16} \end{pmatrix} \quad \mathbf{E}_{2,1} = \begin{pmatrix} e_{33,1} & \cdots & e_{33,16} \\ \vdots & \ddots & \vdots \\ e_{64,1} & \cdots & e_{64,16} \end{pmatrix}$$

(a) Tiling of a 64×16 2D eigenstructure $\mathbf{E}$ into two tiles of 32×16

$$\mathbf{E} = \begin{pmatrix} e_{1,1} & \cdots & e_{1,64} \\ \vdots & \ddots & \vdots \\ e_{32,1} & \cdots & e_{32,64} \\ e_{33,1} & \cdots & e_{33,64} \\ \vdots & \ddots & \vdots \\ e_{64,1} & \cdots & e_{64,64} \end{pmatrix} \quad \mathbf{E}_{1,1} = \begin{pmatrix} e_{1,1} & \cdots & e_{1,32} \\ \vdots & \ddots & \vdots \\ e_{32,1} & \cdots & e_{32,32} \end{pmatrix} \quad \mathbf{E}_{2,2} = \begin{pmatrix} e_{33,33} & \cdots & e_{33,64} \\ \vdots & \ddots & \vdots \\ e_{64,33} & \cdots & e_{64,64} \end{pmatrix}$$

(b) Tiling of a 64×64 3D eigenstructure $\mathbf{E}$ into four tiles of 32×32

Figure 5: *Dividing of 2D and 3D eigenstructures $\mathbf{E}$ into tiles. Tiling the eigenstructures allows for distributing a single eigenstructure across multiple blocks, storing each tile in shared memory of the corresponding block. As 2D eigenstructures have a fixed width of* 16 *columns, they are just split horizontally. 3D eigenstructures are of arbitrary size $N \times M$. They are split in both directions.*

3D eigenstructure matrices into 2 and 4 tiles, respectively. As each block accesses just one tile of the eigenstructure, it does not compute the actual limit point, but an intermediate result. The final limit point is computed by aggregating all intermediate results across the corresponding blocks.

4.2.3. Layered Cells

The limit evaluation of layered cells follows Equation (2) presented in Section 3.2. The 2D eigenstructure is split into tiles as described in the previous section in order to optimize shared memory usage. Each tile of the eigenstructure is assigned to one block, which computes all intermediate limit points produced by the tile. For the thread-based evaluation, each thread computes one or more intermediate limit points. For the warp-based evaluation, the same number of intermediate points is computed by each warp. Figure 6 shows the computation of the layered subdivision basis functions N, which are then multiplied with the individual control points.

In addition to its sub-matrix of the 2D eigenstructure, each block stores the control points of the corresponding layered cell in shared memory to reduce access times. The warp-based kernel also stores the basis functions along each parameter axis in shared memory. To reduce bank conflicts during the evaluation process, the sub-matrices of the eigenstructures are transposed while being copied to shared memory and are therefore stored in column-major order. The implementation supports 2D eigenstructure tiles of 16×16 and

$$\mathbf{b}_u = \begin{pmatrix} b_{u1} \\ b_{u2} \\ b_{u3} \\ b_{u4} \end{pmatrix} \quad \mathbf{b}_v = \begin{pmatrix} b_{v1} \\ b_{v2} \\ b_{v3} \\ b_{v4} \end{pmatrix} \quad \mathbf{b}_w = \begin{pmatrix} b_{w1} \\ b_{w2} \\ b_{w3} \\ b_{w4} \end{pmatrix} \quad \mathbf{b}_{uv} = vec(\mathbf{b}_u \mathbf{b}_v^T)$$

$$\mathbf{N} = \begin{pmatrix} b_{w1}\mathbf{E}\mathbf{b}_{uv} \\ b_{w2}\mathbf{E}\mathbf{b}_{uv} \\ b_{w3}\mathbf{E}\mathbf{b}_{uv} \\ b_{w4}\mathbf{E}\mathbf{b}_{uv} \end{pmatrix} = \begin{pmatrix} b_{w1}\mathbf{E}_{1,1}\mathbf{b}_{uv} \\ b_{w1}\mathbf{E}_{2,1}\mathbf{b}_{uv} \\ b_{w2}\mathbf{E}_{1,1}\mathbf{b}_{uv} \\ b_{w2}\mathbf{E}_{2,1}\mathbf{b}_{uv} \\ b_{w3}\mathbf{E}_{1,1}\mathbf{b}_{uv} \\ b_{w3}\mathbf{E}_{2,1}\mathbf{b}_{uv} \\ b_{w4}\mathbf{E}_{1,1}\mathbf{b}_{uv} \\ b_{w4}\mathbf{E}_{2,1}\mathbf{b}_{uv} \end{pmatrix}$$

$$\text{block}_1 \qquad \text{block}_2$$

$$\mathbf{N}_1 = \begin{pmatrix} b_{w1}\mathbf{E}_{1,1}\mathbf{b}_{uv} \\ b_{w2}\mathbf{E}_{1,1}\mathbf{b}_{uv} \\ b_{w3}\mathbf{E}_{1,1}\mathbf{b}_{uv} \\ b_{w4}\mathbf{E}_{1,1}\mathbf{b}_{uv} \end{pmatrix} \qquad \mathbf{N}_2 = \begin{pmatrix} b_{w1}\mathbf{E}_{2,1}\mathbf{b}_{uv} \\ b_{w2}\mathbf{E}_{2,1}\mathbf{b}_{uv} \\ b_{w3}\mathbf{E}_{2,1}\mathbf{b}_{uv} \\ b_{w4}\mathbf{E}_{2,1}\mathbf{b}_{uv} \end{pmatrix}$$

Figure 6: *Computation of the weight matrix using a tiled 2D eigenstructure $\boldsymbol{E}$ and three univariate B-Spline basis functions $\boldsymbol{b}_u$, $\boldsymbol{b}_v$, $\boldsymbol{b}_w$. As shown in Figure 5, each tile is assigned to one block on the GPU. Each block computes a part of the weight matrix $\boldsymbol{N}$. The intermediate limit points are given by the product of the partial weight matrix and the respective control points. They are accumulated in global memory via atomic operations to compute the actual limit point.*

32×16, implementing one thread-based kernel and one warp-based kernel per tile size – *LAY*16*x*16 and *LAY*32*x*16, respectively.

4.2.4. Irregular Cells

In addition to the 2D eigenstructures, evaluating irregular cells requires the 3D eigenstructures as shown in Equation (3). Our massively-parallel method uses approaches analogous to the evaluation of layered cells described in the previous section. The 3D eigenstructures are tiled and assigned to individual blocks, which compute the corresponding intermediate limit points.

Each block stores its sub-matrix of the 3D eigenstructure, the local control points of the cell, as well as the number of basis functions along each parameter axis u, v, w in shared memory, as this data is constant across all limit points the block computes. To save registers for the thread-based kernel, the size, data pointer, k and n of the 2D eigenstructures are also stored in shared memory, although they are only accessed by a single thread. The warp-based kernel additionally stores the actual basis functions along each parameter axis, the univariate and trivariate basis functions, as this data is accessed by each thread of a warp. Similar to the kernels for evaluating layered cells, the sub-matrices of the 3D eigenstructures are stored in column-major order. Figure 7 shows the computation of the weights for a limit point using a tiled 3D eigenstructure. The 3D eigenstructure tiles are supported in sizes of 32×32 (*IRR*32*x*32) and 64×32 (*IRR*64*x*32). Each tile size again implements a thread-based kernel and a warp-based kernel.

$$\mathbf{b}_u = \begin{pmatrix} b_{u1} \\ b_{u2} \\ b_{u3} \\ b_{u4} \end{pmatrix} \quad \mathbf{b}_v = \begin{pmatrix} b_{v1} \\ b_{v2} \\ b_{v3} \\ b_{v4} \end{pmatrix} \quad \mathbf{b}_w = \begin{pmatrix} b_{w1} \\ b_{w2} \\ b_{w3} \\ b_{w4} \end{pmatrix}$$

$$\mathbf{b}_{uvw} = vec(vec(\mathbf{b}_u \mathbf{b}_v^T)\mathbf{b}_w^T)$$

$$\mathbf{N} = \mathbf{E}\mathbf{b}_{uvw} = \begin{pmatrix} \begin{pmatrix} \mathbf{E}_{1,1} & \mathbf{E}_{1,2} \end{pmatrix} \mathbf{b}_{uvw} \\ \begin{pmatrix} \mathbf{E}_{2,1} & \mathbf{E}_{2,2} \end{pmatrix} \mathbf{b}_{uvw} \end{pmatrix}$$

$$\text{block}_1 \qquad \text{block}_2$$

$$\mathbf{N}_1 = \mathbf{E}_{1,1} \begin{pmatrix} \mathbf{b}_{uvw1} \\ \vdots \\ \mathbf{b}_{uvw32} \end{pmatrix} + \mathbf{E}_{1,2} \begin{pmatrix} \mathbf{b}_{uvw32} \\ \vdots \\ \mathbf{b}_{uvw64} \end{pmatrix}$$

$$\text{block}_3 \qquad \text{block}_4$$

$$\mathbf{N}_2 = \mathbf{E}_{2,1} \begin{pmatrix} \mathbf{b}_{uvw1} \\ \vdots \\ \mathbf{b}_{uvw32} \end{pmatrix} + \mathbf{E}_{2,2} \begin{pmatrix} \mathbf{b}_{uvw32} \\ \vdots \\ \mathbf{b}_{uvw64} \end{pmatrix}$$

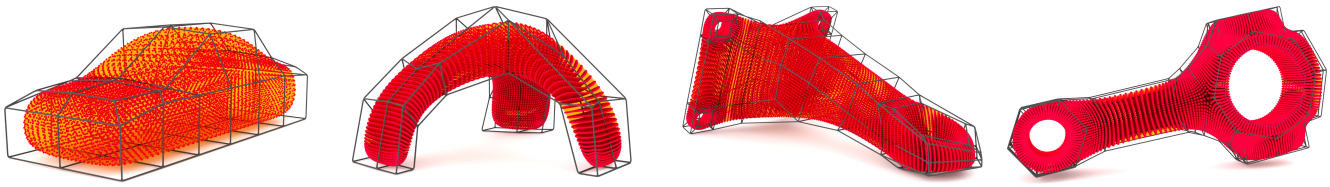
Figure 7: *Computation of the weight matrix using a tiled 3D eigenstructure $\boldsymbol{E}$ (see Figure 5) and three univariate B-Spline basis functions $\boldsymbol{b}_u$, $\boldsymbol{b}_v$, $\boldsymbol{b}_w$. Each block processes one tile of the eigenstructure and computes part of the weight matrix $\boldsymbol{N}$. Similarly to layered evaluation, the intermediate limit points are accumulated in global memory via atomic operations.*

5. Results

To assess the performance of our GPU-parallel limit evaluation method, we measured the computation time and geometric error for four CC-solid models with increasing topological complexity – i.e. increasing numbers of EEs, EVs and crease edges. The four models *car*, *tripod*, *engine mount* and *connecting rod* are shown in Figure 8 while their topology is analyzed in Table 1. For each of the four models and each cell class, we evaluated various numbers of limit points on various subdivision levels with our GPU-parallel approach on three different NVIDIA GPUs – GeForce GTX 1070, RTX 2080 Ti, and Quadro GP100 – as well as with a CPU-parallel implementation of Altenhofen et al.'s original algorithm [AMW+18] on an Intel i7-6700k CPU. The subset of chosen GPUs represents a mid-range and a high-range consumer GPU, as well as a GPU for professional use featuring HBM2 memory.

5.1. Thread-Based vs. Warp-Based Evaluation

First, we compared the computation times of the thread-based and the warp-based kernels for the different cell classes (see Section 4.2). As the thread-based kernels focus on arithmetic operations, they outperform the warp-based kernels when evaluating a sufficiently large number of limit points per cell. In these situations, the many sample points spawn enough threads to properly utilize the GPU, while also benefitting from the higher arithmetic



(a) Car model (b) Tripod model (c) Engine mount model (d) Connecting rod model

Figure 8: *Visualization of the four CC-solid models used for benchmarking our approach. The images show the control mesh in gray, the limit surface transparently in yellow and evaluated limit points in red.*

Model	n	#cells	Regular		Layered		Irregular	
			64	48	16x16	32x16	32x32	64x32
Car	0	24	0 (-)	0 (-)	16 (66.7%)	0 (-)	8 (33.3%)	0 (-)
	1	192	40 (20.8%)	88 (45.8%)	56 (29.2%)	0 (-)	8 (4.2%)	0 (-)
	2	1536	792 (51.6%)	600 (39.1%)	136 (8.9%)	0 (-)	8 (0.5%)	0 (-)
Tripod	0	1088	360 (33.1%)	504 (46.3%)	168 (15.4%)	20 (1.8%)	12 (1.1%)	24 (2.2%)
	1	8704	5448 (62.6%)	2776 (31.9%)	380 (4.4%)	40 (0.5%)	12 (0.1%)	48 (0.6%)
	2	69632	55944 (80.3%)	12696 (18.2%)	804 (1.2%)	80 (0.1%)	12 (0.0%)	96 (0.1%)
Engine Mount	0	3120	1088 (34.9%)	1288 (41.3%)	352 (11.3%)	140 (4.5%)	0 (-)	252 (8.1%)
	1	24960	16288 (65.3%)	7136 (28.6%)	752 (3.0%)	280 (1.1%)	0 (-)	504 (2.0%)
	2	199680	164144 (82.2%)	32416 (16.2%)	1552 (0.8%)	560 (0.3%)	0 (-)	1008 (0.5%)
Conn. Rod	0	9056	4336 (47.9%)	2940 (32.5%)	708 (7.8%)	352 (3.9%)	164 (1.8%)	556 (6.1%)
	1	72448	53968 (74.5%)	14700 (20.3%)	1476 (2.0%)	928 (1.3%)	180 (0.2%)	1196 (1.7%)
	2	579584	507184 (87.5%)	64620 (11.1%)	3012 (0.5%)	2080 (0.4%)	212 (0.0%)	2476 (0.4%)

Table 1: *Topology of the four test models for different subdivision levels n. The cells are categorized into different classes and sub-classes, each evaluated by a specific compute kernel. The sub-classes for regular cells are defined by the number of control points – 64 for internal cells and 48 for boundary cells – while the sub-classes of layered and irregular cells are defined by the size of the eigenstructure tiles. For every cell class, the table shows the absolute and relative number of cells in the corresponding CC-solid model.*

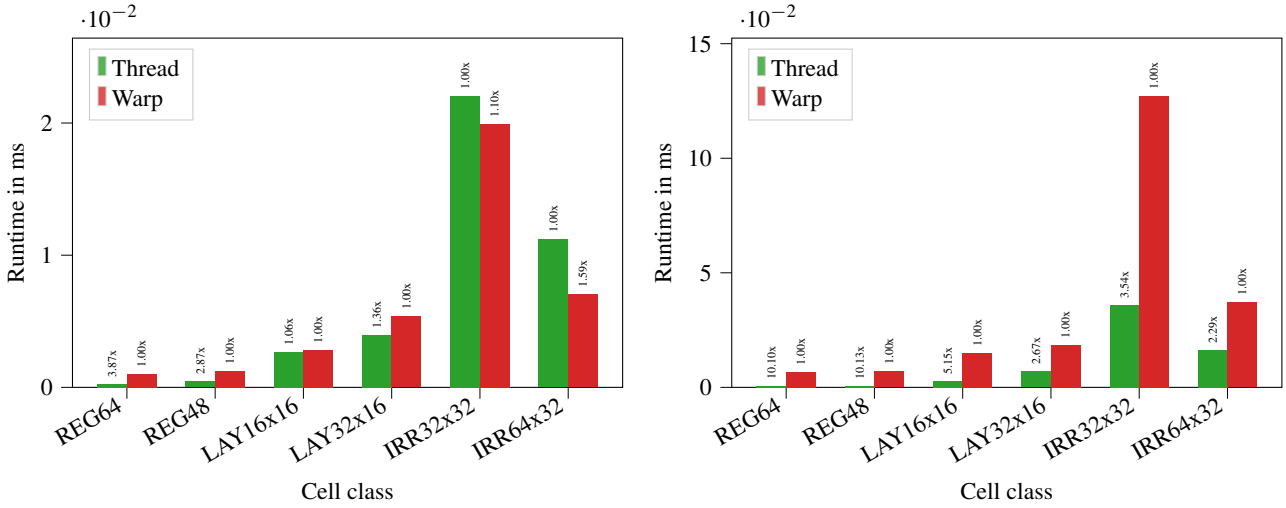
intensity of the thread-based kernel. The warp-based kernels, on the other hand, can still maintain proper GPU utilization when evaluating only a few sample points per cell due to their inherently higher thread count per block.

Using $4^3 = 64$ and $8^3 = 512$ sample points per cell (see Figure 9), the thread-based kernel outperforms the warp-based kernel for regular and layered cells. Even though the thread-based approach is faster for these cell classes independent of the sample count, the speedups of the thread-based approach over the warp-based approach are only measured as 3.87× ($REG64$), 2.87× ($REG48$), 1.06× ($LAY16x16$) and 1.36× ($LAY32x16$), respectively, when sampling each cell at 64 points. The higher sample count of 512 sample points leads to higher speedups of 10.10×, 10.13×, 5.15× and 2.67×, respectively. For irregular cells, the low number of sample points per eigenstructure tile when sampling at 64 sample points prohibits the thread-based kernel from fully utilizing the GPU. The warp-based kernel shows speedups of 1.10× and 1.59× over the thread-based approach. Increasing the sample count to 512 also increases the number of threads per block and enables the thread-based kernel to outperform the warp-based kernel even for irregular cells by factors of 3.54× and 2.29×.

The effect of using the *hybrid* approach of dynamically choosing the optimal kernel based on the evaluation parameters can be seen in Figure 10 where we measured the performance when evaluating entire models instead of individual cell classes. The hybrid approach achieves a speedup of 1.63× on the engine mount model and 1.56× on the connecting rod, compared to pure warp-based evaluation. For both models, the thread-based approach outperforms the warp-based approach, but is slower than the hybrid approach, showing speedups of 1.40× and 1.31× over the warp-based kernels. The tripod shows the thread-based approach to be the fastest, with a speedup of 1.15× over the warp-based approach, while the hybrid approach only achieves a speedup of 1.10×. The car model features too few cells to produce conclusive results.

5.2. CPU vs. GPU

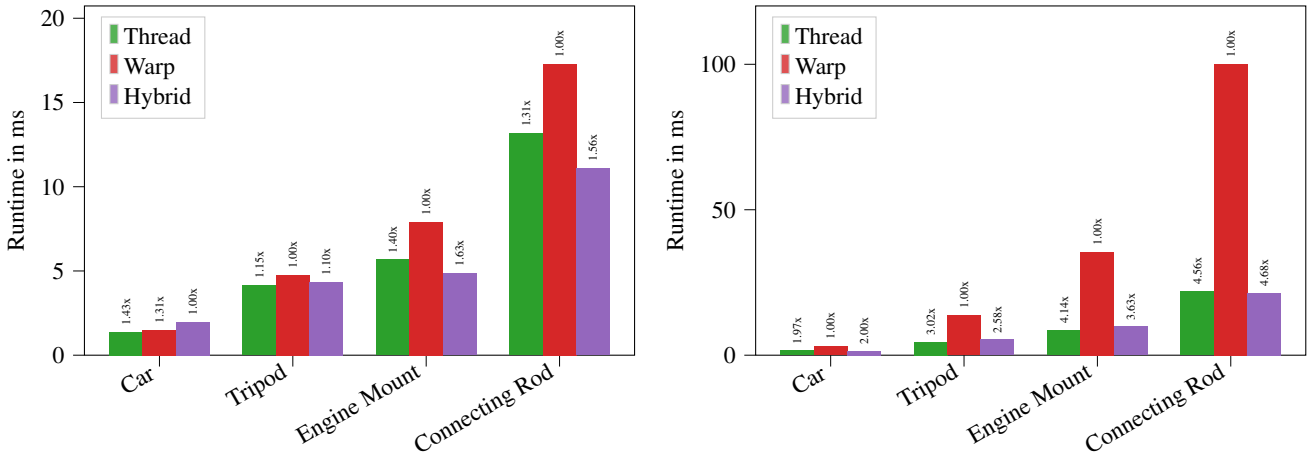
In this section, we present the performance comparison of our GPU-parallel technique and the CPU-parallel implementation of Altenhofen et al.'s approach [AMW+18]. Figure 11 shows the runtime and speedup per cell class for evaluating 64 and 512 sample points per cell using the Intel Core i7 CPU as well as our set of different NVIDIA GPUs. For all cell classes, all GPUs evaluate the cells in less time than the CPU. The data show that compute power is the most important factor influencing the evaluation time. The RTX 2080 Ti outperforms the other GPUs in most cases. Only the



(a) Connecting rod, evaluating $4^3 = 64$ sample points per cell

(b) Connecting rod, evaluating $8^3 = 512$ sample points per cell

Figure 9: *Average evaluation time of the thread-based and warp-based kernels per cell for every cell class. The speedup of each kernel type compared to the slower kernel type is shown above each bar. The connecting rod model is the most complex in our benchmark, featuring all cell classes. The warp-based kernels outperform the thread-based kernels when evaluating irregular cells with* 64 *sample points per cell. For all other configurations, the thread-based kernels evaluate the cells in less time.*



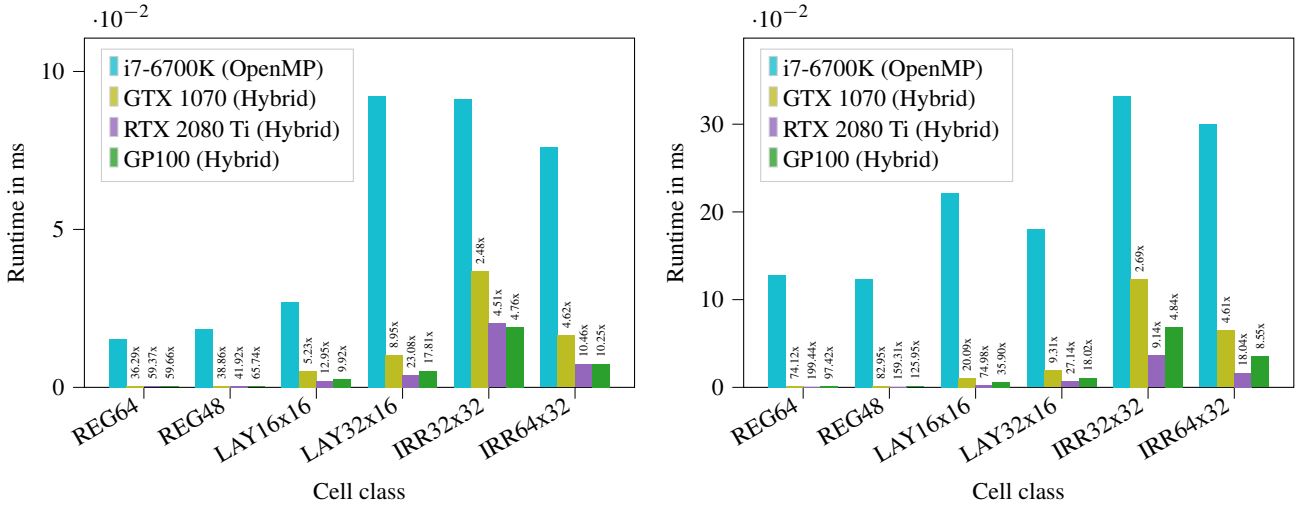
(a) $4^3 = 64$ sample points per cell

(b) $8^3 = 512$ sample points per cell

Figure 10: *Runtime for each of the four models for evaluating the entire model using either the thread-based, warp-based or hybrid approach. The speedup of each kernel type over the slowest kernel type is shown above each bar. For* 64 *samples per cell, the hybrid approach exploits the speedup of the warp-based approach when evaluating irregular cells (see also Figure 9(a)), while for* 512 *samples, the thread-based approach is dominant.*

irregular cells of the connecting rod, sampled with 64 samples per cell, show little difference between the RTX 2080 Ti and GP100.

However, the margin between the RTX 2080 Ti and the Quadro GP100 is larger than the difference in raw compute power of the two GPUs. The extensive use of atomics for the evaluation of layered and irregular cells limits memory bandwidth during writes to global memory and therefore prohibits the GP100 from making use of its wide memory bus. Our method excels at evaluating regular cells. The RTX 2080 Ti produces speedups of 199.44× (*REG*64) and 159.31× (*REG*48) compared to the CPU when sampling 512 sample points per cell. The more complex cell classes show lower speedups of 74.98× (*LAY*16*x*16) and 27.14× (*LAY*32*x*16) for layered and 9.14× (*IRR*32*x*32) and 18.04× (*IRR*64*x*32) for irregular cells. Reducing the sample count also reduces parallelism of the



(a) Connecting rod, evaluating $4^3 = 64$ sample points per cell

(b) Connecting rod, evaluating $8^3 = 512$ sample points per cell

Figure 11: *Average evaluation time per cell using the CPU and three different GPUs, evaluating* 64 *and* 512 *sample points per cell. The speedup achieved on each GPU compared to the CPU implementation is shown above each bar. Increasing the number of sample points also increases the speedup of our approach, especially for regular cells. For irregular cells, the RTX 2080 Ti and GP100 show little difference in the runtime of limit evaluation for* 64 *samples, while the increased sample count of* 512*, allows the RTX 2080 Ti to outperform the GP100 significantly. As in Figure 9, we only show the results for the connecting rod in these diagrams.*

limit evaluation and directly impacts performance. The speedups are lowered to 59.37×, 41.92×, 12.95×, 23.08×, 4.51× and 10.46×, respectively.

The number of cells per cell class also influences the performance of the algorithm. Cell classes containing only a few cells spawn an insufficient number of blocks to fully utilize the GPUs and therefore limit the speedup of the GPU approach compared to the CPU implementation. The speedups achieved for evaluating layered and irregular cell classes with a large number of cells – *LAY*32*x*16 and *IRR*32*x*32 – in the connecting rod are much higher (especially for the RTX 2080 Ti) than the speedups for *LAY*32*x*16 and *IRR*32*x*32. The RTX 2080 Ti is affected by this the most as it features the most SMs. The performance of the GTX 1070 is impacted less, as this GPU features fewer SMs but a larger number of FP32 units per SM [NVI20b, NVI20a].

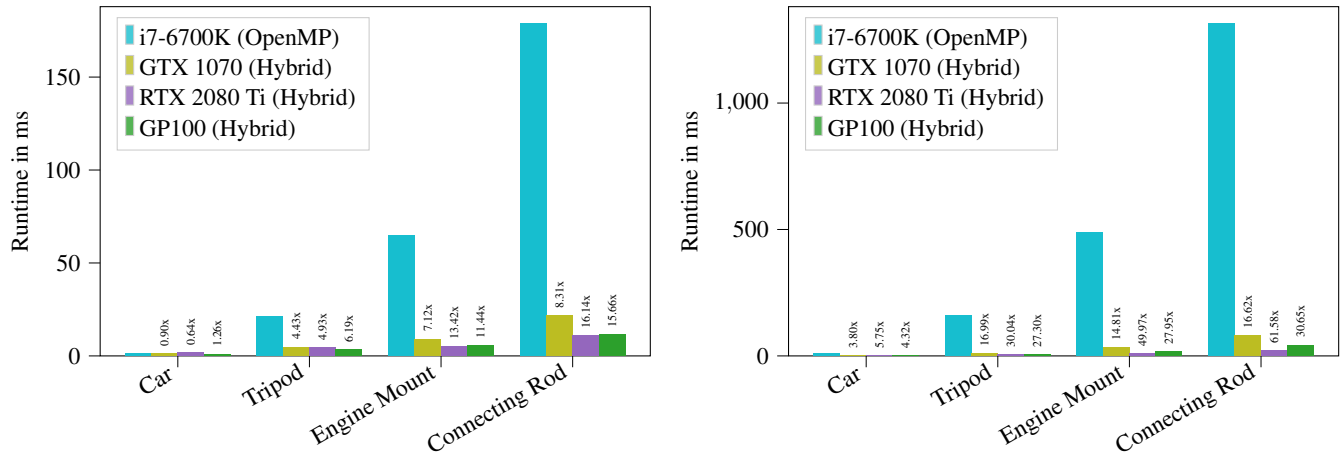
Figure 12 shows the performance measurements for evaluating entire models. As for the individual cell classes, the overall evaluation time of the models is primarily determined by the processing power of the GPU, as the RTX 2080 Ti shows the lowest runtimes, followed by the GP100 and the GTX 1070. The achieved speedups depend on the total number of cells, as the car model only achieves a maximum speedup of 5.75×, while the tripod, engine mount and connecting rod models achieve maximum speedups of 30.04×, 49.97× and 61.58×.

5.3. Sample Points per Cell vs. Number of Cells

Like Altenhofen et al.'s original approach, our GPU-parallel evaluation method evaluates every single limit point in constant time. Linearly increasing the sample count also linearly increases the computations per cell. Therefore, in a sequential environment, the runtime of the algorithm is expected to be linear w.r.t. the sample count. In the massively parallel environment, the increased work does not automatically increase the runtime by the same amount, as the algorithm can use more threads per block to compute the additional limit points. Increasing the sample count will therefore not significantly impact throughput of the GPU-parallel implementation as long as the number of samples is less than or equal to the maximum block size. If the sample count exceeds the maximum block size, the work per thread increases linearly, as each thread computes multiple limit points. The setup time of the block, as well as copying data from global memory to shared memory, however, is independent of the sample count. An increase in the number of limit points evaluated per block will therefore reduce the overhead relative to the evaluation time and lead to sublinear growth of the runtime w.r.t. the number of sample points, as shown in Figure 13.

In addition to the number of sample points per cell, the performance benefits of our method depends on the total number of cells and the number of cells in specific cell classes. Small models with too few cells – also in individual classes – do not scale well with the massively parallel paradigms of modern GPUs (see Section 5.2).

As seen in Figure 11, the GPU-parallel implementation is most efficient when evaluating regular cells. The GPU-parallel implementation will therefore benefit from subdividing models with a high number of irregular cells, as just one of the resulting 8 cells will remain irregular while the other 7 will either have a regular or layered topology. Simultaneously reducing the sample count by a factor of two along each axis produces the same limit points, while increasing the ratio of regular to irregular cells. Figure 14 shows that subdividing the model before evaluation improves the runtime



(a) $4^3 = 64$ sample points per cell

(b) Models at $8^3 = 512$ sample points per cell

Figure 12: *Runtime per model for the CPU and three different GPUs, all evaluating* 64 *and* 512 *sample points per cell. The speedup for each GPU compared to the CPU is shown above each bar. Increasing the sample count also increases the speedups of our approach compared to the CPU implementation due to the increased thread count per block and therefore higher parallelism. Sampling the car model with* 64 *samples per cell results in a slowdown due to poor utilization of the GPU and the overheads of memory transfer and kernel launches negate the faster evaluation times of the massively parallel implementation.*

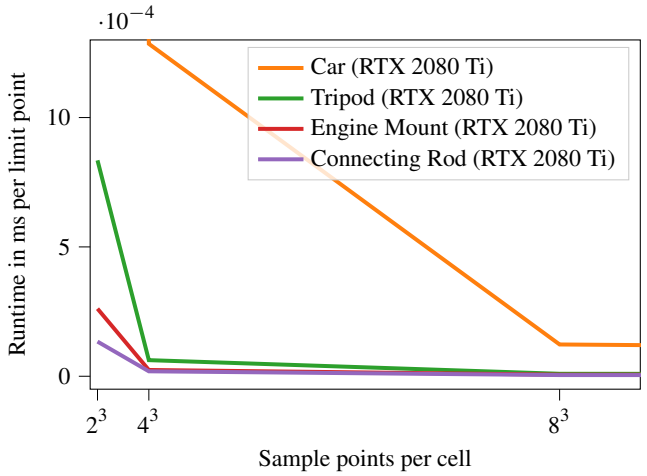


Figure 13: *Runtime per limit point for increasing number of sample points per cell. As every sample point is evaluated in constant time, the average runtime per limit point would be constant for a sequential algorithm. In a massively parallel environment, however, the runtime per sample point decreases with higher sample counts, due to higher GPU utilization and parallelism.*

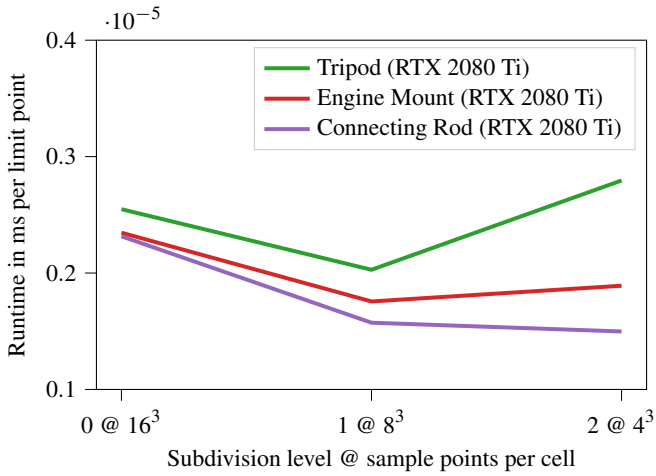


Figure 14: *Comparing the runtime per limit point for different subdivision levels and sample counts. For each explicit subdivision of the model, the sample count is reduced by a factor of 8, resulting in the same number of total limit points. The car model has been omitted in this diagram as the model features too few irregular cells to benefit from subdivision.*

up to a certain level, until too few sample points per cell remain in order to utilize the GPU efficiently. For the tripod and the engine mount, the best results were achieved at one subdivision step and 8^3 sample points. The connecting rod shows the smallest runtime at two subdivision steps and 4^3 sample points (Figure 14).

5.4. Accuracy

In order to maximize the computational throughput on consumer GPUs with a limited number of FP64 cores, our GPU-parallel method uses single precision floating point arithmetic. While this

produces less accurate results than the CPU-parallel implementation, which uses double precision floating point arithmetic, it allows for a high performance on a wider range of hardware.

To analyze the difference in accuracy, we computed the average error err_{avg} over all limit points for each model and compared the results for the thread-based and the warp-based kernels. Using the CPU implementation as the baseline, the error err of a single limit point $p(u,v,w)$ is defined by the Euclidean distance between the limit point evaluated on the CPU and the limit point evaluated on the GPU:

$$err(p(u,v,w)) = \|p_{cpu}(u,v,w) - p_{gpu}(u,v,w)\|_2 \qquad (4)$$

As this computes the absolute error of each point, the error will scale with the maximum distance of the limit points to the origin.

Due to the different kernel designs, the error depends on whether the thread-based or the warp-based evaluation approach is used to evaluate the limit points. The warp-based kernel uses significantly fewer add operations per limit point compared to the thread-based kernel by computing every 32nd weight and aggregating the results via warp-shuffle reduction. Since each add operation can introduce numerical errors due to floating point arithmetic, the limit points evaluated with the warp-based kernels expectedly show a higher accuracy than the ones of the thread-based kernels.

For evaluating the car model at 512 sample points per cell, a purely thread-based evaluation produced an average error of $err_{avg} = 1.10 \cdot 10^{-7}$ compared to the CPU implementation, while a purely warp-based approach produced an error of $7.63 \cdot 10^{-8}$. Evaluating the engine mount, tripod and connecting rod models resulted in errors of $7.49 \cdot 10^{-6}$ vs. $4.64 \cdot 10^{-6}$, $2.80 \cdot 10^{-7}$ vs. $1.75 \cdot 10^{-7}$, $1.59 \cdot 10^{-5}$ vs. $9.94 \cdot 10^{-6}$ – thread-based vs. warp-based. As the hybrid evaluation technique dynamically chooses between both kernel types, the errors lie between these values. As all CUDA capable GPUs follow the same standards for floating point arithmetic [NVI20a], the error is consistent across all tested GPUs.

6. Conclusions and Future Work

In our paper, we presented the first massively parallel approach for evaluating the limit volume of Catmull-Clark solid models. Adapting the concepts of Altenhofen et al.'s constant-time limit evaluation technique [AMW+18], we designed specific compute kernels for the individual cell classes, to minimize resource usage while maximizing performance. To satisfy the need for a high number of threads, we introduced thread-based as well as warp-based evaluation kernels that differ in the workload of each thread. Furthermore, we split the computation of the limit of layered and irregular cells across multiple blocks by tiling the 2D and 3D eigenstructures.

We analyzed the performance of our massively parallel algorithm considering the complexity of the CC-model to be evaluated and the number of sample points per cell. The test models showed speedups of up to 199.44× for regular cells, 74.98× for layered and 18.04× for irregular cells compared to the CPU-parallel implementation resulting in total speedups of between 3.80× and 61.58× for our four test models on three different GPUs.

Due to the fact, that each cell or eigenstructure sub-matrix spawns one block on the GPU, the algorithm shows better scaling for a large number of cells with few sample points, than for a small number of cells with a large number of sample points. This also leads to low speedups compared to the CPU implementation when evaluating models with few cells, as the GPU cannot be fully utilized, leaving whole SMs idle. The worst case – low cell and sample count – spawns a small number of blocks and warps, which reduces the potential for latency hiding. To counteract this, our method features a hybrid approach, dynamically choosing between the thread-based and the warp-based kernels to increase parallelism by spawning an entire warp – instead of one thread – per limit point.

Our approach can be further improved by using CUDA streams. Overlaying memory transfers and kernel execution would reduce the evaluation times of the CC-solid models, while the concurrent execution of multiple kernels would reduce the impact of underpopulated cell classes.

So far, we focused on evaluating the positions $(x,y,z) \in \mathbb{R}^3$ of the CC-solid limit points. In the future, we plan to apply our massively parallel approach to isogeometric analysis as well as to slicing multi-material CC-solid models in order to measure the performance gains in these concrete applications.

References

[AESF21] Christian Altenhofen, Tobias Ewald, André Stork, and Dieter W. Fellner. Analyzing and improving the parameterization quality of catmull–clark solids for isogeometric analysis. *IEEE Computer Graphics and Applications*, 41(3):34–47, 2021. 1, 3

[ALG+18] Christian Altenhofen, Thu Huong Luu, Tim Grasser, Marco Dennstädt, Johannes Sebastian Mueller-Roemer, Daniel Weber, and André Stork. Continuous property gradation for multi-material 3d-printed objects. In *Solid Freeform Fabrication Symposium*, volume 29, pages 1675–1685, 2018. 1

[AMW+18] Christian Altenhofen, Joel Müller, Daniel Weber, André Stork, and Dieter W. Fellner. Direct limit volumes: Constant-time limit evaluation for catmull-clark solids. In Hongbo Fu, Abhijeet Ghosh, and Johannes Kopf, editors, *Pacific Graphics Short Papers*. The Eurographics Association, 2018. 1, 2, 3, 6, 7, 11

[ASSF17] Christian Altenhofen, Felix Schuwirth, André Stork, and Dieter Fellner. Implicit mesh generation using volumetric subdivision. In Fabrice Jaillet and Florence Zara, editors, *Workshop on Virtual Reality Interaction and Physical Simulation*, VRIPHYS'17, Lyon, France, 2017. 1

[BHU10] Daniel Burkhart, Bernd Hamann, and Georg Umlauf. Isogeometric finite element analysis based on Catmull-Clark subdivision solids. *Computer Graphics Forum*, 29 (5):1575–1584, 2010. 1

[BS02] Jeffrey Bolz and Peter Schröder. Rapid evaluation of Catmull-Clark subdivision surfaces. In *Proceedings of the seventh international conference on 3D Web technology*, pages 11–17. ACM, 2002. 2

[BSWX02] Chandrajit Bajaj, Scott Schaefer, Joe Warren, and Guoliang Xu. A subdivision scheme for hexahedral meshes. *The visual computer*, 18(5-6):343–356, 2002. 2

[CC78] Edwin Catmull and James Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-aided design*, 10(6):350–355, 1978. 2

[CQ03] Yu-Sung Chang and Hong Qin. Mass: Multiresolutional adaptive solid subdivision. 05 2003. 2

[DSHB09] Tor Dokken, Vibeke Skytt, Jochen Haenisch, and Kjell Bengtsson. Isogeometric representation and analysis: bridging the gap

between cad and analysis. In *47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*, page 1172, 2009. 2

[EDE17] Ben Ezair, Daniel Dikovsky, and Gershon Elber. Fabricating functionally graded material objects using trimmed trivariate volumetric representations. In *Proceedings of SMI'2017 Fabrication and Sculpting Event (FASE), Berkeley, CA, USA*, 2017. 2

[HCB05] Thomas J.R. Hughes, John A. Cottrell, and Yuri Bazilevs. Isogeometric analysis: Cad, finite elements, nurbs, exact geometry and mesh refinement. *Computer methods in applied mechanics and engineering*, 194(39):4135–4195, 2005. 2

[JM99] Kenneth I. Joy and Ron MacCracken. The refinement rules for catmull-clark solids. In *Technical. Report*. Citeseer, 1999. 1, 2

[LAE+19] Thu Huong Luu, Christian Altenhofen, Tobias Ewald, André Stork, and Dieter Fellner. Efficient slicing of catmull–clark solids for 3d printed objects with functionally graded material. *Computers & graphics*, 82:295–303, 2019. 1, 2

[Loo87] Charles T. Loop. *Smooth Subdivision Surfaces Based on Triangles*. PhD thesis, Department of Mathematics, The University of Utah, Masters Thesis, 01 1987. 2

[LZC+18] Heng Liu, Paul Zhang, Edward Chien, Justin Solomon, and David Bommes. Singularity-constrained octahedral fields for hexahedral meshing. *ACM Transactions on Graphics (TOG)*, 37(4):93, 2018. 2, 3

[MAS17] Johannes Sebastian Mueller-Roemer, Christian Altenhofen, and André Stork. Ternary sparse matrix representation for volumetric mesh subdivision and processing on GPUs. *Computer Graphics Forum*, 36(5):59–69, 2017. 2

[ME16] Fady Massarwi and Gershon Elber. A b-spline based framework for volumetric object modeling. *Computer-Aided Design*, 78:36–47, 2016. 2

[NLMD12] M. Nießner, C. Loop, M. Meyer, and T. DeRose. Feature-adaptive gpu rendering of catmull-clark subdivision surfaces. *ACM Transactions on Graphics (TOG)*, 31(1):6, 2012. 2

[NVI20a] NVIDIA Corporation. CUDA C++ PROGRAMMING GUIDE, 2020. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. 4, 9, 11

[NVI20b] NVIDIA Corporation. CUDA GPUS | NVIDIA Developer, 2020. https://developer.nvidia.com/cuda-gpus. 9

[NVI21a] NVIDIA Corporation. CUDA Zone | NVIDIA Developer, 2021. https://developer.nvidia.com/cuda-zone. 1, 4

[NVI21b] NVIDIA Corporation. OpenCL | NVIDIA Developer, 2021. https://developer.nvidia.com/opencl. 1

[SHW04] Scott Schaefer, Jan Hakenberg, and J. Warren. Smooth subdivision of tetrahedral meshes. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 147–154. ACM, 2004. 2

[Sta98a] Jos Stam. Exact evaluation of Catmull-Clark subdivision surfaces at arbitrary parameter values. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 395–404, New York, NY, USA, 1998. ACM, ACM. 1, 2, 3

[Sta98b] Jos Stam. Fast evaluation of loop triangular subdivision surfaces at arbitrary parameter values. In *Computer Graphics (SIGGRAPH'98 Proceedings, CD-ROM Supplement)*, 1998. 2

[XXD+20] Jin Xie, Jinlan Xu, Zhenyu Dong, Gang Xu, Chongyang Deng, Bernard Mourrain, and Yongjie Jessica Zhang. Interpolatory catmull-clark volumetric subdivision over unstructured hexahedral meshes for modeling and simulation applications. *Computer Aided Geometric Design*, page 101867, 2020. 1