Revisiting The Vertex Cache: Understanding and Optimizing Vertex Processing on the modern GPU

BERNHARD KERBL, MICHAEL KENZEL, ELENA IVANCHENKO, DIETER SCHMALSTIEG, and MARKUS STEINBERGER, Graz University of Technology

In this paper, we question the premise that graphics hardware uses a post-transform cache to avoid redundant vertex shader invocations. A large body of existing work on optimizing indexed triangle sets for rendering speed is based upon this widely-accepted assumption. We conclusively show that this assumption does not hold up on modern graphics hardware. We design and conduct experiments that demonstrate the behavior of current hardware of all major vendors to be inconsistent with the presence of a common post-transform cache. Our results strongly suggest that modern hardware rather relies on a batch-based approach, most likely for reasons of scalability. A more thorough investigation based on these initial experiments allows us to partially uncover the actual strategies implemented on graphics processors today. We reevaluate existing mesh optimization algorithms in light of these new findings and present a new mesh optimization algorithm designed from the ground up to target architectures that rely on batch-based vertex reuse. In an extensive evaluation, we measure and compare the real-world performance of various optimization algorithms on modern hardware. Our results show that some established algorithms still perform well. However, if the batching strategy of the target architecture is known, our approach can significantly outperform these previous state-of-the-art methods.

CCS Concepts: • **Computing methodologies → Graphics processors**; *Massively parallel algorithms*;

Additional Key Words and Phrases: Mesh Optimization, Vertex Processing, GPU, Post-transform Cache

ACM Reference Format:
Bernhard Kerbl, Michael Kenzel, Elena Ivanchenko, Dieter Schmalstieg, and Markus Steinberger. 2018. Revisiting The Vertex Cache: Understanding and Optimizing Vertex Processing on the modern GPU. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2, Article 29 (August 2018), 16 pages. https://doi.org/10.1145/3233302

1 INTRODUCTION

The same transformed vertex is on average required six times during rendering of a typical triangle mesh. To avoid redundant vertex transformation, graphics processors have traditionally employed a *post-transform cache*, often simply referred to as the vertex cache. The optimization of rendering sequences to maximize locality of vertex references and, thus, increase rendering performance given such hardware is a well-researched area. The resulting mesh optimization algorithms have seen widespread adoption and are routinely found in art asset pipelines for real-time rendering.

However, public information on the actual implementation details in graphics hardware is scarce, driving researchers to adopting micro-benchmarks to enable understanding of crucial hardware behavior [Barczak 2016; Jia et al. 2018]. The lack of mention of a concept as well-established as the post-transform cache in some of the more recent articles [Kubisch 2015; Purcell 2010] raises the question to which degree many widely-accepted assumptions concerning efficient vertex processing still align with the reality of modern graphics processors. The prevailing assumption that modern

Authors' address: Bernhard Kerbl, bernhard.kerbl@icg.tugraz.at; Michael Kenzel, michael.kenzel@icg.tugraz.at; Elena Ivanchenko, elena.ivanchenko@icg.tugraz.at; Dieter Schmalstieg, schmalstieg@tugraz.at; Markus Steinberger, steinberger@icg.tugraz.at, Graz University of Technology.

graphics processors employ a simple, centralized vertex cache has previously been challenged by others in the computer graphics community [Barczak 2016; Giesen 2011]. However, no detailed analysis beyond faint suspicions has been published on this matter.

In this work, we put the hypothesis that graphics hardware is using a post-transform cache to the test. We design and conduct experiments that clearly show that the behavior of vertex processing on a modern graphics processing unit (GPU) is not consistent with the presence of a traditional post-transform cache. As demonstrated in [Kenzel et al. 2018], a batch based approach seems to more closely match the behavior of modern GPUs, which is not surprising considering that a central cache would likely become an obstacle to scalability as devices grow more and more parallel. Through thorough analysis, we are able to reveal some of the details of reuse strategies which current GPUs from three major contemporary manufacturers seem to be following. Armed with the knowledge gathered in these reverse-engineering attempts, we are able to predict the way in which a specific GPU will process a given input stream with great accuracy. We then turn our attention towards the topic of mesh optimization and outline a general algorithmic framework for optimizing indexed triangle sets for batch-based vertex processing, given that the concrete batching routine used by the target hardware is known. Finally, we present an extensive evaluation of the real-world performance of previous mesh optimization algorithms as well as our own approach.

2 RELATED WORK

The idea of exploiting vertices being shared by neighboring triangles to remove redundancy and thereby reduce vertex processing load is not new. Early attempts approached the issue as a problem of compressing and decompressing the input geometry, which led to the conception of triangle strips as well as algorithms that turn arbitrary meshes into a stripified representation. [Chow 1997; Deering 1995; Evans et al. 1996]

Hoppe [1999] introduced the idea of relying on a post-transform cache in the graphics processor to dynamically take advantage of locality of reference when possible, without requiring all geometry submitted for rendering to be encoded first. They also presented their now widely-known mesh processing algorithm to generate cache-optimized triangle strips from input geometry data. In order to maximize the cache hit rate, triangles are reordered by greedily growing strips and introducing cuts when they become too long to effectively exploit the cache during their processing. An implementation of Hoppe's algorithm was included as part of the D3DX library from DirectX 9 onward and has since been migrated to be part of the *DirectXMesh* library. Its popularity and effectiveness gave rise to several other, conceptually similar optimization methods. The specifics of cache operations is thereby often abstracted to avoid sensitivity to differences in hardware architectures. Nevertheless, most algorithms assume either a first-in-first-out (FIFO) or least-recently-used (LRU) cache and aim to minimize the average cache miss rate (ACMR) by reordering the input vertex indices according to their spatial and topological traits. Details on cache behavior of early GPU architectures can be found in corresponding documentation [Riguer 2006], as well as hardware proposals and simulation frameworks [Sheaffer et al. 2004; Wang et al. 2011].

Generalizing from triangle strips to arbitrary indexed triangles sets has allowed more recent work to consider highly elaborate reordering schemes. Lin and Yu [2006] describe the *K-Cache* algorithm, which iteratively selects focus vertices and subsequently outputs all connected faces, before marking the focus vertex as visited. The selection of an adequate focus vertex includes a predictive simulation of the cache evolution in each iteration, which renders their approach quite time-consuming. Forsyth [2006] presents a linear-speed mesh optimization algorithm that employs a similar metric, but assumes an exponential falloff for the probability of a cache hit. Hence, it omits predictive simulation for final cache positions and requires no parameterization. As a consequence,

low ACMR values comparable to those of Lin and Yu can be achieved at a significantly lower preprocessing cost. Its low runtime make the algorithm a popular choice for processing by industry.

Sander et al. [2007] have presented an extremely fast, scalable algorithm named *Tipsify* for reordering vertex indices as part of their mesh optimization tool chain. Their considerations concerning overdraw can also be extended to animated scenes [Han and Sander 2016]. *Tipsify* works particularly well for larger cache sizes, as has been shown by experimental simulations. Its low cost and intuitive nature led to the inclusion of *Tipsify* as part of AMD's Triangle Order Optimization Tool (*Tootle*), which provides an approachable implementation for end users.

Recent work on out-of-core geometry processing has shown, that the fact that data can be rearranged without changing the mesh itself can be exploited to yield a successful strategy for improving performance even in applications beyond rendering [Isenburg and Lindstrom 2005].

Cache considerations may not only be applied when rendering geometry, but also when processing or transmitting geometry. Chhugani and Kumar [2007] have explored the concept of cache-based optimization to achieve a compression-friendly topology representation. The authors report cache miss rates on par with those of Lin and Yu, as well as extremely low storage requirements of only ~8 bits per triangle. Considering the entire cache hierarchy from hard drive to main memory to rendering system, appropriately reordering geometry data can significantly increase transfer [Tchiboukdjian et al. 2008, 2010; Yoon and Lindstrom 2007].

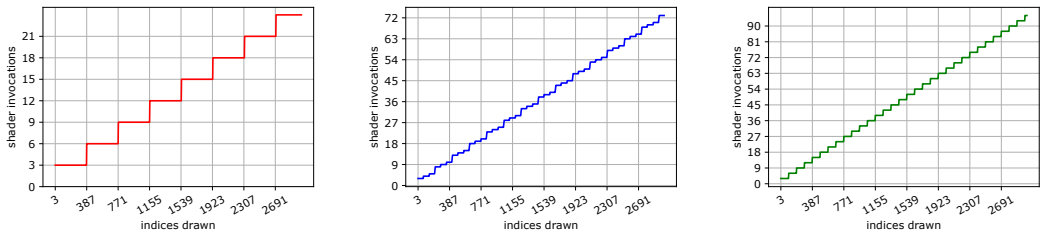
While cache simulations are usually applied when reordering meshes, a fast alternative is to use space-filling curves on top of the mesh to create locality between triangles [Vo et al. 2012]. Although the achieved cache hit rates are ~5% below simulation-based approaches, the high efficiency and reduced storage requirements can be favorable under highly constrained execution environments.

3 GPU VERTEX REUSE STRATEGIES

To go about answering the question whether a modern GPU uses a traditional vertex cache or not, we conduct the following experiment: We set up a vertex buffer holding three vertices and an index buffer containing the sequence $\{0, 1, 2, 0, 1, 2, 0, \ldots\}$ up to a given maximum length. We then draw an increasingly larger portion of this sequence. Using a Direct3D 11 pipeline statistics query, we record the number of times the vertex shader is invoked when drawing each portion. To account for effects from the size of transformed vertex data, we repeat the experiment for different numbers of output vertex attributes, ranging from 0 to the maximum 124 supported by Direct3D.

Fig. 1 shows the behavior of the vertex shader invocation count as the number of indices drawn increases for a selection of GPUs, which are representative of the behavior we have observed for different vendors. Assuming the presence of a common post-transform cache of reasonable size, for this input stream, we would expect each vertex to be transformed exactly once. However, on almost all GPUs that we tested, we observe the vertex invocation count rising continuously with the stream length passing certain thresholds. The only exception we have seen stem from older Intel GPUs, which did in fact exhibit the behavior expected of a post-transform cache. This is consistent with documentation publicly available for these models [Intel Corporation 2013].

To summarize our findings so far: None of the assessed modern GPU models employ a central post-transform cache. The number of output vertex attributes does not seem to affect any GPU's ability to take advantage of vertex reuse. Furthermore, we can also rule out the presence of multiple post-transform caches instead of a single shared one, as such an architecture would lead the vertex invocation count to eventually plateau. However, even when drawing up to 30 million triangles, the vertex invocation count kept consistently rising on all modern GPUs.



(a) AMD Radeon R7 260X (b) Intel HD Graphics 520 (c) NVIDIA GeForce GTX 1080 Ti

Fig. 1. The number of times the vertex shader is invoked on a selection of GPUs of different vendors when drawing a recurring 0, 1, 2 index sequence of increasing length. A post-transform cache of reasonable size would achieve perfect reuse in this scenario, processing every vertex exactly once and leading to an overall number of three shader invocations independently of input size. The observed "staircasing" is indicative of a batch-based approach for processing of the input primitive stream where vertex reuse is only considered locally within the same batch.

3.1 Measuring vertex reuse

To quantify vertex processing efficiency, we use the *average shading rate* (ASR), *i.e.*, the average number of vertex shader invocations per triangle drawn. A similar measure that has been commonly used in the past is the *average cache miss rate* (ACMR). However, in light of our findings, we would argue that thinking about vertex reuse in terms of caching is not necessarily beneficial.

In order to measure the ASR achieved on a particular GPU for a given triangle mesh, we can use the same approach as before where we relied on a Direct3D 11 pipeline statistics query that allows us to measure the total number of vertex shader invocations. On more recent hardware, a more fine-grained result can be obtained by using atomic operations to increment a per-vertex counter from within the vertex shader.

3.2 Collecting detailed batching information

Computation on modern GPUs is organized around parallel lock-step execution of small groups of logical threads in the lanes of *single instruction multiple data* (SIMD) cores. These smallest thread groups that form the units of SIMD-wide parallel computation are typically referred to as *warps*, *wavefronts*, or simply *waves*.

In order to gain further insight as to how vertex processing is orchestrated on a modern GPU, our next step is to collect more detailed information about which vertices are processed together in the same wavefront. Luckily, experimental support for Shader Model 6.0 (recently added to the Direct3D 12 API) exposes wave-level operations that allow cross-communication between threads sharing a wavefront. Although not officially supported for vertex shading, we found that they can in fact be used even then, at least on all GPUs with Direct3D 12 that we tested. Based on these instructions, we are able to write a vertex shader that outputs the information of exactly which vertex indices were processed in parallel within the same wave. We start by determining how many threads are currently running the vertex shader in a given wave. The thread with the lowest lane id then atomically increments a counter to allocate space in an output buffer and communicates the start index to all other threads in the wave. Using a parallel prefix sum, we assign a unique index to each active thread. Finally, each thread writes the vertex index they are currently assigned to process into the output buffer at the given offset. The first thread additionally outputs the number of vertex indices contained within the recorded batch.

3.3 Identifying batch patterns and boundaries

We describe here a set of reproducible experiments that we used to confirm the trends and patterns we observed from the collected batch information. Their description also serves to outline the logic and constraints for forming batches on NVIDIA and AMD GPUs from recent generations with practical examples. The results from our initial assessment give us a starting point to further investigate batching behavior: The locations at which steps for the vertex invocation count occur actually give us the first indicator, namely the maximum size of batches in an ideal case (*i.e.*, perfect reuse throughout the mesh). For the sake of brevity, we will refer to this parameter henceforth as *MAX_SIZE*. For NVIDIA, we can consistently observe steps every 96 indices (see Figure 1c). This makes sense, in so much as we know warps on NVIDIA to run 32 threads in lockstep. Thus, in an ideal scenario, a batch size of 96 indices allows every thread to process a separate triangle, consisting of 3 indices. On the AMD R7 200, we find the *MAX_SIZE* to be 384 (see Figure 1a). It should be mentioned that, on the RX Vega 56, this figure seems to vary, relative to the number of indices in the index buffer, peaking at 384 as well. We have not investigated this behavior further, as even the smallest of our test cases triggers the maximum batch size of 384 to take effect. Finally, we confirm these numbers by drawing from an index buffer where each assumed batch is filled with a different, repeating triplet of unique indices. For a *MAX_SIZE* of 6, *e.g.*, the index buffer would be constructed as $\{0, 1, 2, 0, 1, 2 \mid 3, 4, 5, 3, 4, 5 \mid 6, 7, \ldots\}$. On Intel models HD 520 and HD 630 we found that vertex reuse seems to be more dynamic, largely depending on the number of different indices being rendered. For the repeating test sequence of 3 unique indices, full reuse only happens within 66 indices and partial reuse up to 208 (see Figure 1c); however, for other data we have found a maximum reuse window of up to 1791 indices.

Next, we vary the data within each assumed batch of length *MAX_SIZE*, to see if other limitations apply. For this, we can fill the index buffer with a consistently increasing sequence of N indices, where N is divisible by 3 and $N < MAX\_SIZE$, followed by $(MAX\_SIZE - N)$ indices in triplets $\{N-4, N-3, N-2\}$. Simply put, the index buffer contains $\frac{N}{3}$ unique triangles and then repeats the second-but-last triangle until we hit *MAX_SIZE*. Assuming that vertex reuse in a batch works at least over two consecutive triangles, the average shading rate must therefore remain at $\frac{N \cdot 3}{MAX\_SIZE}$ as long as the batch goes on. On NVIDIA, we found that this condition failed at $N = 33$ and the ASR jumped to $\frac{(N+1) \cdot 3}{MAX\_SIZE}$. We found this number to be curiously close to the NVIDIA warp size. Further variations based on this procedure have indeed confirmed that one batch may reference no more than 32 unique vertices. Formulating it as a rule to apply to batch formation, we refer to the figure of maximum allowed unique vertices as *MAX_UNIQUES*. We also noticed that occurrences of later ASR steps were shifted according to batches being terminated early due to the *MAX_UNIQUES* constraint. Hence, it follows that NVIDIA GPUs can dynamically choose start and end positions for portions of the index buffer that qualify as a batch where reuse applies. In contrast, the identified boundary *MAX_SIZE* always applies on AMD GPUs without fail. There is no value for *MAX_UNIQUES* that causes irregularities in development of the ASR. For generality and notational convenience, we can therefore define that $MAX\_UNIQUES = MAX\_SIZE$ on AMD. For Intel, we observed *MAX_UNIQUES* to equal 128, which has been reported on Haswell before [Barczak 2016]. For 128 different indices, $MAX\_SIZE = 1791$ can also be observed. It seems that Intel's behavior can be described as *MAX_SIZE* being proportional to the number of different indices already being found, introducing batch boundaries depending on how many different indices have been recorded. A closer analysis of the hardware behavior also indicated that Intel internally completely sorts all indices and assigns them in alternating orders to 8-wide SIMD execution units.

Finally, we need to determine if and when the reusability of vertex information inside a batch can expire. We begin by considering the separating distance between indices as a potential factor.

We start from our default repeating index buffer $\{0, 1, 2, 0, 1, 2, 0, \ldots\}$ of length *MAX_SIZE* and replace one entry at location i with a new index $X > 2$. Another instance of X will be placed at a distance n from the first, at position $i + n$. Increasing n towards the limit *MAX_SIZE*, a step in the ASR will then indicate that this particular vertex could not be reused over that distance. On NVIDIA GPUs, we found that this is the case for $n = 41$. We noted a slight difference in the general retention behavior, depending on the API being used. On OpenGL, the identification of reusable vertices is always equivalent to a look-back of distance 42 into the index buffer. With DirectX, this behavior changes to a variable 42±3 look-back, depending on the local index within the a triangle, *i.e.*, the index buffer location modulo 3. Specifically, reuse is detected between two positions in the index buffer i and j ($i < j$) iff they reference the same vertex and their distance is d:

$$\begin{aligned} d \leq 44, &\quad \text{if } i \equiv 0 (\text{mod } 3) \\ d \leq 40 \vee d = 42, &\quad \text{if } i \equiv 1 (\text{mod } 3) \\ d \leq 42 \wedge d \neq 40, &\quad \text{if } i \equiv 2 (\text{mod } 3) \end{aligned}$$

In terms of data structures, the retention can thus be modeled as a FIFO cache of length ~42 that always pushes queried entries, regardless of whether it already contains them or not. Any index that is not found in the retention model will count towards the number of unique vertices. On AMD, we could not identify a simple limiting separation distance within a batch. Instead, we tested whether the number of *unique* indices separating two instances of X would have an effect on the ASR. In fact, we found that reuse inside a batch on AMD does not work for two indices that are separated by more than 14 unique indices. Hence, the retention model of AMD for vertex reuse inside a batch can be expressed by an LRU cache of size 15. In contrast to NVIDIA, this "forgetful" behavior does not influence how batches are formed, however it will obviously affect the ASR, since "forgotten" vertices that reoccur need to be shaded once more. On Intel, we could not identify a predictable condition for expiration of previously computed vertices above the already stated relationship between *MAX_SIZE* and *MAX_UNIQUES*. Interestingly, we could observe certain vertices being shaded more often than others for no apparent reason, *e.g.*, for the repetitive sequence $\{0, 1, 2, 0, 1, 2, \ldots\}$, vertex 2 is shaded three times as often as 0 and 1, whereas for a sequence $\{0, 1, 2, 3, 0, 1, 2, 3, \ldots\}$ all vertices are shaded equally and overall fewer shader invocations occur than in the first case. Based on these observations, it seems impossible to capture Intel's retention model with a simple construct. Thus, assuming the best reuse scenario with sufficiently many different indices being rendered, we approximate Intel with a FIFO cache of size *MAX_UNIQUES*.

Although we were able to identify these peculiarities in the architectures and verified them using the batch information produced according to Section 3.2, we can still observe unresolved artifacts in the batching behavior. For NVIDIA, we found that the FIFO reuse does not hold if indices cross over a multiple of 2^{16}, *i.e.*, 16 bit boundary. For instance, if two indices I_a and I_b are placed in a batch where $\lfloor I_a/2^{16} \rfloor \neq \lfloor I_b/2^{16} \rfloor$, reuse according to the look-back retention may not occur. A potential explanation is an internal optimization for processing 16-bit wide indices, which can be handled more efficiently. For AMD, we found that the LRU size is also not always 15, but might sometimes be slightly longer or shorter. However, we were unable to find a comprehensive model to reproduce these exceptions. Analyzing the information of concurrently processed indices by wavefronts, we found that parts of different batches might be forwarded to the same wavefront on AMD, which ensures near perfect occupancy of the compute units. We assume that this combination can be modeled as a separate step after creating batches and identifying reuse within a batch with the sole goal of avoiding idle threads.

3.4 Predicting batch breakdown for the GPU

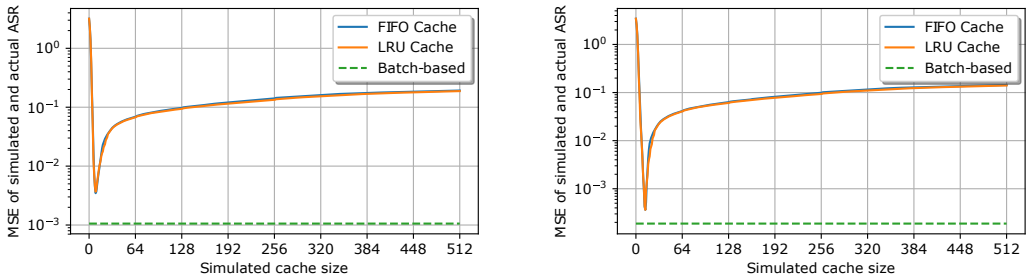
Given the information that we obtained by means described in Section 3.2 and interpreted in Section 3.3, we are able to make a qualified prediction about how a full list of triangle indices is split into separate batches on the GPU. Our base approach for predicting the batch breakdown is listed in Algorithm 1. Note that the properties that we identified for individual architectures can be abstracted by allowing the parameterization of the *MemoryModule*, as well as the boundary values *MAX_SIZE* and *MAX_UNIQUES*. The *MemoryModule* encapsulates the behavior of how previously shaded data is maintained for quick reuse. *MAX_SIZE* and *MAX_UNIQUES* define the maximum allowed number of indices in a batch and of unique vertices used, respectively. The algorithm processes the index buffer in groups of three, assuming the input to be a triangle mesh. Shaded and available vertices (*uniques*) are tracked by the algorithm, as well as the total number of indices (*size*) in the current batch. If one of the boundary conditions is detected (line 16), we terminate the current batch and start a new one. This includes resetting the memory for available vertex information and—if a termination condition is detected mid-triangle—reloading its already visited indices (lines 22 to 27).

We evaluate the accuracy of our prediction scheme by simulating the batch breakdown for our full test suite on GPUs available to us. NVIDIA-style batching, as discerned above, can be achieved by setting *MAX_SIZE* to 96 and *MAX_UNIQUES* to 32. Availability of vertex data is tracked by the *MemoryModule*. For lack of a more accurate solution, we use a 15-wide LRU cache on AMD here and precise retention on NVIDIA, according to the description in Section 3.3. As AMD shows no limitation of unique vertices for a batch, we set both *MAX_SIZE* and *MAX_UNIQUES* to the hard boundary of 384 indices. This effectively takes care of the first stage of AMD's two-tiered batching process. Although the second tier of AMD's approach influences the assignment and combination of vertices to wavefronts, it does not affect batching boundaries, nor the number of shader invocations. Thus, for the purpose of predicting or improving vertex reuse, the second stage can simply be ignored.

As a guideline for accuracy, we use the number of invocations returned by each simulation to compute the ASR. We compare our results to global FIFO/LRU cache-based simulations of variable size. This is equivalent to the optimization scenarios used *e.g.* by Hoppe [1999] and Lin and Yu [2006], which use look-ahead simulations to minimize the predicted ASR. Assessing the cache simulation accuracy at various sizes serves two relevant purposes: First, in order to disprove a centralized post-transform vertex cache, we need to ensure that no reasonably-sized cache can produce the ASR values we measured. Since we make no assumptions about the size of such a fictitious structure, we need to check a wide range of different sizes in order to conclusively rule out centralized caching. Second, it serves as an insight into which cache sizes actually come closest to the actual experienced behavior. This information may guide researchers and developers to select the appropriate sizes, if they choose to continue using cache-based approaches. As will be shown, it also allows us to pick the best possible configuration for cache-based algorithms to compare against in our evaluation. We omit this step for Intel models, since Intel's behavior cannot be captured by our batch-based approach based on what we have learned from hardware analysis. The mean squared error (MSE) to the actual measured ASR on the GPU (averaged over our full test suite) is shown in Figures 2a and 2b. Evidently, our batch-based approach is significantly more accurate than cache-based simulations. At their best configuration (*i.e.*, with a cache size of 10/15), the MSE of FIFO and LRU variants is still more than 3× that of ours on NVIDIA, and 2× on AMD. In fact, our prediction for batches and ASR on NVIDIA are completely accurate for models that contain fewer than 2^{16} indices. The remaining reported error stems from our inability to predict handling of larger scenes. As mentioned above, NVIDIA appears to employ an (as of yet unidentified) mechanism

Algorithm 1: Predicting batch breakdown

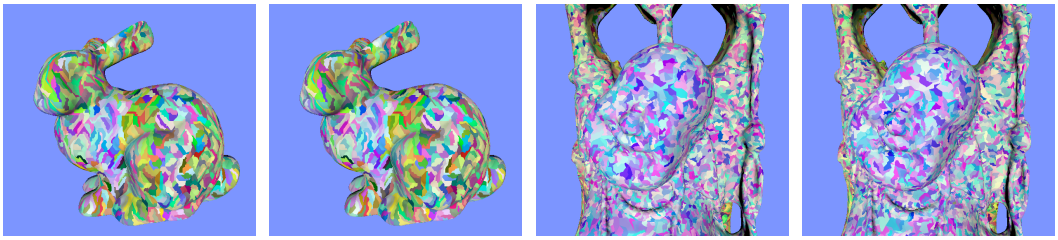
```
1  in unsigned int Indices[]
2  out unsigned int invocations ← 0
   // Initialize
3  unsigned int pos ← 0
4  unsigned int size ← 0
5  unsigned int uniques ← 0
6  MemoryModule Available ← ∅
7  for each triangle △ in the index buffer Indices do
8      added ← 0
9      Available′ ← Available
10     for i ← 0 to 2 do
11         id ← ith index △(i) of triangle △
12         if {id} ⊄ Available′ then
13             added ← added + 1
14         else
15             Notify Available′ of id at (pos + i)
16         if (uniques + added) > MAX_UNIQUES or (size + i) > MAX_SIZE then
17             Found batch in Indices from (pos − size) to pos
18             Reset Available, initialize its contents to ∅
19             Available′ ← Available
20             size ← 0
21             uniques ← 0
               // Continue with current triangle △, load its i indices into
                  Available′
22             for j ← 0 to i do
23                 if △(j) ⊄ Available′ then
24                     Store △(j) in Available′
25                     added ← added +1
26                 else
27                     Notify Available′ of △(j) at (pos + j) .
28         else
29             Store id in Available′
30     Available ← Available′
31     pos ← pos + 3
32     size ← size + 3
33     uniques ← uniques + added
34     invocations ← invocations + added
```



(a) Cache-based vs ours on NVIDIA

(b) Cache-based vs ours on AMD (RX Vega 56)

Fig. 2. Accuracy of shading rate predictions using different simulation techniques. For NVIDIA, the MSE with our batch-based approach is less than a third, for AMD it is half of the closest cache-based simulation. On NVIDIA, the minimum occurs at a cache size of 10 for FIFO, on AMD at 15 for LRU. Although based on a centralized cache instead of multiple parallel ones, these results seem to confirm our assumption for retention on the respective architectures.



(a) Bunny prediction (b) Bunny measured (c) Buddha prediction (d) Buddha measured

Fig. 3. Rendering of the predicted and actually reported batches as they are processed on NVIDIA for the bunny and happy_buddha models. Random colors are used to distinguish batch boundaries. For scenes with a vertex count $< 2^{16}$, such as bunny, we can accurately predict how batches are formed (a,b). For larger scenes, our model is not exact, but can still predict ASR and overall batch generation more accurately than cache-based simulations (c,d).

that avoids reuse of indices lower and higher than 2^{16} in the same batch. We visualize the effect of this property on our batch prediction in Figure 3. On AMD, the slightly higher residual error is due to behavior not fully captured by our prediction, including the deviation of the actual size used for LRU retention, as discussed in Section 3.3.

4 BATCH-BASED MESH OPTIMIZATION

In order to determine whether knowledge of batch boundaries can be beneficial for preprocessing algorithms, we design a simple and thus efficient mesh reordering algorithm based on the knowledge we gathered from the hardware behavior. Focusing on low runtime, our algorithm follows a greedy strategy that continuously grows batches by choosing an origin and iteratively adding faces to the batch. Similarly to existing approaches, the selection of the next face to be "added" to the mesh in each step is governed by a cost or priority function that identifies the most suitable candidate. The entire algorithm and its integration with the batch prediction function is outlined in Algorithm 2.

Our algorithm considers four parts for the priority of each face: *Vertex Reuse R*, *Vertex Valence V*, *Face Distance D* and *Batch Neighborhood N*. The priority p for each triangle $\triangle$ is computed by

Algorithm 2: Batch-based Mesh Optimization

```
in VertexInformation vI[]
in unsigned int Indices_orig[]
out unsigned int Indices_opt[]
BatchingPredictor bp

Set reuseVertices ← ∅
Map distance ← ∅
PriorityQueue queue ← ∅
for each triangle △ ∈ Indices_orig do
    V(△) ← Σ_{i=0}^{2} vI[△(i)].valence
    Insert △ with score V(△) · k_v into queue
while queue ≠ ∅ do
    △ ← highest-scoring entry in queue
    if can add △ according to bp then
        Add △ to Indices_opt
        Erase △ from queue
        // add R(·) and reduce V(·), add D(·) according to new △
        for i ← 0 to 2 do
            r_i ← 0 if △'(i) ∈ reuseVertices else 1
            for △' ∈ vI[△(i)].faces do
                adjust p(△') by (r_i · k_r − k_v)
        for △' ∈ neighbors of △ do
            D_{△,△'} ← shortestPath(△, △')
            adjust p(△') by k_d · D_{△,△'}
            distance(△') ← k_d · D_{△,△'}
        Add indices of △ to reuseVertices
        // update R(·) for 'forgotten' vertices
        for v ∈ reuseVertices do
            if v can no longer be reused according to bp then
                remove v from reuseVertices
                for △' ∈ vI[v].faces do
                    adjust p(△') by −k_r
    else
        // remove R(·) due to batch boundary, remove D(·) and add N(·)
        for v ∈ reuseVertices do
            for △' ∈ vI[v].faces do
                adjust p(△') by −k_r
        for △' ∈ neighbors of △ do
            adjust p(△') by k_n − distance(△')
        Reset bp for new batch
        reuseVertices ← ∅
        distance ← ∅
```

weighing the four factors with appropriate coefficients:

$$p(\triangle) = k_r \cdot R(\triangle) + k_v \cdot V(\triangle) + k_d \cdot D(\triangle) + k_n \cdot N(\triangle). \quad (1)$$

- *Vertex Reuse* captures the number of vertices of a triangle that will result in a reuse of the vertex shading information when the triangle is added to the current batch, *i.e.*, how many of its vertices are already in the batch and will be identified for reuse when this triangle is added. Obviously, choosing a high priority k_r is important to build batches that result in as few vertex shader invocations as possible. To determine how many vertices of a triangle actually will be reused, it is essential to know how and when the hardware introduces batch boundaries and how it identifies reusable vertex information.
- *Vertex Valence* corresponds to the sum over all vertex valences of the triangle, whereby whenever a triangle is added to the index buffer, the valence for all its vertices is reduced by one. One can think of this as making a copy M_{copy} of the input mesh on which the vertex valence is tracked and whenever a triangle is added to the index buffer, it is removed from the mesh copy. Using a negative priority k_v will prioritize triangles with low valence, *i.e.*, those which will likely have a low chance of leading to reuse if they are left over. This approach also guarantees that the algorithm will start batches from mesh boundaries (and from boundaries of previous batches) and not randomly grow isolated batches all over the mesh.
- *Face Distance* is computed on the dual graph of the mesh and corresponds to the sum over all distances from all faces currently in the batch to all surrounding faces, *i.e.*, triangles that can be reached from all triangles in the batch over few other triangles show a small D. This means that a negative k_d will prioritize those triangles that can be reached easily, and will thus lead to more "circular" batches. On the other hand, a positive k_d will lead to elongated batches. Especially under the assumption that vertex reuse is possible among all triangles within a batch, "circular" batches intuitively are preferable as they will show the highest potential reuse.
- *Batch Neighborhood* is 1 for triangles that are direct neighbors to triangles which have been added to completed batches, *i.e.*, are part of the boundary of M_{copy} but are not part of the boundary in the original mesh. Thus, N allows to guide batches along already existing batches (positive k_n) or slightly push batches away from already existing ones (negative k_n). The former helps to avoid fragmentation of batches; the latter helps to avoid elongated batches, especially when $k_d = 0$.

In order to increase the probability of a cache hit, previous work usually considers the immediate neighborhood of the last processed triangle first, thereby catering to the presumed, underlying cache architecture. In contrast, we argue that the order of faces within a batch is irrelevant for mesh optimization, as long as the hardware can identify that the same vertices are referenced. Furthermore, knowing when the hardware inserts a batch boundary, allows an optimization algorithm additional freedom, as there is no possibility for the next triangle to create any reuse with previously referenced vertices. Thus, the optimization algorithm can "jump" to any other mesh location. Especially for a hardware that inserts batch boundaries often, *e.g.*, NVIDIA, these jumps have a significant impact on the overall performance. In the mindset of previous algorithms, they can be seen as very frequent cache resets.

Computing and updating these factors for all triangles whenever a triangle is added to the current batch or a batch is completed, continuously changes the priority of all potential triangles. Using a priority queue with low update complexity when the priorities are increased or decreased is essential for efficient processing. To this end, we employ a Fibonacci Heap as our priority queue.

Empiric assessment has shown that the configuration

$$k_c = 1024, k_v = -4, k_d = -1, k_n = -1 \tag{2}$$

yields consistently good results. In terms of computational effort, distance information D is by far the most expensive factor of our cost function, since its maintenance requires updates to all faces that are neighbors to the current batch. In order to reduce processing time while maintaining low average shading rate, we can eliminate D from the cost function and set $k_v = k_n = -3$.

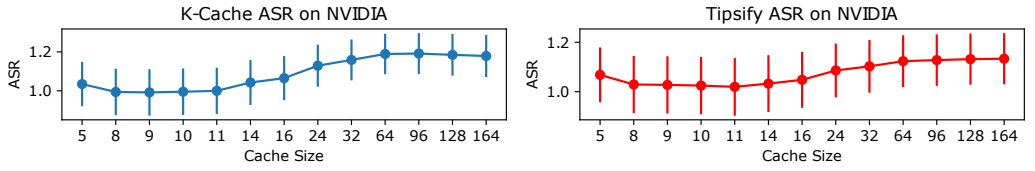
The previously discussed algorithm is based on some simplified views on the hardware. As mentioned above, we have found that indices that go across multiples of 2^{16} on NVIDIA will prohibit reuse within a batch. Thus, we added a simple additional step that operates on the preprocessed mesh, to make sure that the simulated batch boundaries are not destroyed by this behavior. Whenever we determine (based on our simulation) that a batch contains indices that cross a 16-bit boundary, we duplicate conflicting vertices and add them to the end of the vertex buffer. Obviously, this can again result in the batch crossing over 16-bit boundaries (if the previously largest index is not in the same 16-bit boundary as the new vertex buffer size), which may require us to copy another set of vertices to the back. Instead of referencing the original lower vertices, the indices are then recast to reference the newly added ones. Note, that this step increases the vertex count and thus reduces the *ideal* reuse potential across the entire mesh. However, it will result in better *achieved* reuse within batches and thus improve ASR on NVIDIA.

5 EVALUATION

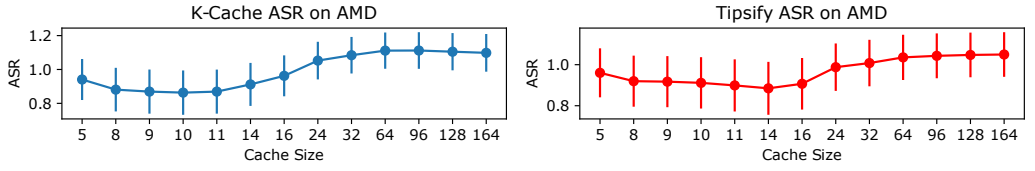
To compare with previous work, we use readily available libraries for algorithms by Hoppe [1999] and Sander et al. [2007] from *DirectXMesh* and *Tootle*, respectively. For the work by Forsyth [2006], we used the standalone version, provided by Adrian Stone, as referenced in the original publication. Regarding K-Cache optimization [2006], we consider our own, faithful C/C++ implementations as representatives of their mesh optimization method. For all techniques, we record the actual ASR yielded by different GPU models, as described in section 3.1. Hence, these results serve not only as a baseline to compare against, but also as a survey on the effectiveness of previous approaches on modern hardware. For input data, we include a variety of commonly used test scenes from the graphics community, as well as clip-space geometry that we captured from five recent video games and an NVIDIA technical demo: Age of Mythology (abbreviated am), Assassin's Creed: Black Flag (as), Deus Ex: Human Revolution (dx), Stone Giant animation (sg), Total War: Shogun 2 (sh), Rise of the Tomb Raider (tr), and The Witcher 3 (tw). All models were rendered as indexed triangle lists using 32-bit indices. For smaller models, we have also tested 16-bit indices and found that this does not influence the resulting ASR in any way.

To enable a fair comparison, we selected the cache sizes for K-Cache and Tipsify that produced the best results on those devices (see Figure 4). The lowest shading rate with K-Cache was achieved with 10 entries on NVIDIA and AMD. For Tipsify, we used a cache size of 11 on NVIDIA, and 14 on AMD. This is in accordance with our previous assessment regarding cache-based ASR prediction: given that these algorithms simulate and optimize the ASR based on the existence of a cache, they perform best at the configuration where a cache-based prediction is closest to the actual behavior, *i.e.*, at a cache size between 10 and 16. On Intel, as previously assumed, the best value for cache size equals its *MAX_UNIQUES* figure (128).

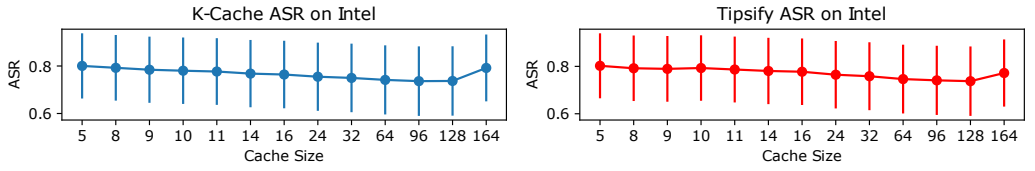
For NVIDIA GPUs, we tested all techniques on the Geforce GTX 780Ti, 980Ti, 1060, 1080 and 1080Ti, covering three different hardware generations (Kepler, Maxwell and Pascal). Recorded shading rates were identical in all cases, regardless of the particular model used, and are shown in Table 1. For reference, we also give the ideal ASR values, *i.e.*, shading rates that would be achieved



(a) NVIDIA (Kepler, Maxwell and Pascal)



(b) AMD (RX Vega 56)



(c) Intel (HD 630)

Fig. 4. We evaluate the ASR of meshes processed with differently parameterized optimization methods K-Cache and Tipsify on NVIDIA, AMD and Intel hardware. Similar to our prediction models, NVIDIA and AMD achieve best performance with cache-based optimizations at sizes between 10 and 16. On Intel, as expected, best reuse occurs at a cache size of 128, corresponding to the maximum number of retained vertices.

with absolute reuse of shaded vertex output. As documented by these results, our simple, batch-based algorithm yields the lowest shading rate out of all contestants in every test scenario but one. For large, carefully crafted meshes, the benefit of our algorithm over the best contender is significant: for the Happy Buddha statue, we report a 15% improvement over the best alternative, K-Cache. For the dragon model, this figure jumps to a 25% improvement over the best result yielded from previous work by Hoppe. For weakly structured meshes, such as the tree model and scenes obtained from captured video game frames, the improvement of the ASR is smaller, but usually still significant in the 3-10% range. A large portion of this improvement is hinged on our post-processing step, which makes sure indices within a batch will not cross 16-bit boundaries. As shown above, we cannot exactly predict NVIDIA's batching for arbitrary input models that contain more than 2^{16} unique vertices (see Figure 3). Note that ignoring this property would mean that the algorithm assumes reuse within batches that do not align with those constructed by the GPU, and virtually all potential benefit is annulled. For game scenes, optimizing without this post-processing step usually results in a 1-10% increase of the ASR. In contrast, for `happy_buddha` and `xyz_dragon`, ignoring 16-bit boundary constraints yields a much higher ASR of 1.01 and 1.15, respectively.

For AMD, we considered an R7 200 from the GCN2 generation, as well as an RX Vega 56. For the AMD models, the results of batching were much less rewarding than for NVIDIA. Although our batch-based algorithm is on par with more elaborate techniques, it is usually bested by either K-Cache and TomF. We note here once more that the identification of the AMD batching functions

Table 1. Achieved ASR with different mesh optimization techniques applied, on recent NVIDIA generations. For all cases but one, our simple, batch-based mesh optimization can outperform elaborate methods, most prominently for the happy_buddha and xyz_dragon test cases. For reference, we also include the ASR that would be possible with perfect reuse (*e.g.*, using a centralized cache of infinite size).

	Hoppe	K-Cache	Tipsify	TomF	Batching	*Perfect Reuse*
sphere	0.83	0.82	0.83	0.88	**0.81**	*0.50*
bunny	0.84	0.84	0.86	0.88	**0.82**	*0.50*
happy_buddha	0.98	0.95	0.98	0.99	**0.81**	*0.50*
xyz_dragon	1.07	1.10	1.08	1.16	**0.82**	*0.50*
tree	2.07	2.09	2.09	2.08	**2.06**	*2.06*
am01	0.97	0.86	0.88	0.87	**0.84**	*0.60*
am02	0.95	0.81	0.81	0.81	**0.78**	*0.48*
as01	0.87	0.85	0.88	0.86	**0.83**	*0.59*
as02	1.27	1.26	1.28	1.26	**1.24**	*1.11*
dx01	0.88	**0.85**	0.90	0.85	0.89	*0.61*
dx02	0.87	0.84	0.88	0.85	**0.84**	*0.62*
sg01	0.87	0.83	0.88	0.84	**0.83**	*0.53*
sg02	0.89	0.85	0.89	0.85	**0.84**	*0.56*
sh01	1.01	0.97	1.00	0.97	**0.92**	*0.74*
sh02	0.98	0.95	0.98	0.95	**0.94**	*0.74*
tr01	0.95	0.89	0.93	0.89	**0.87**	*0.68*
tr02	0.93	0.89	0.92	0.89	**0.88**	*0.66*
tw01	0.87	0.87	0.89	0.89	**0.84**	*0.55*
tw02	1.43	1.39	1.41	1.39	**1.37**	*1.23*

is likely incomplete. However, the properties of the AMD architecture also reduce the impact that correct batching can have. Given the combination of large batches and small cache size, elaborate algorithms focusing on cache optimization achieve ASR rates that are already close to ideal. While fully understanding and incorporating knowledge about batch boundaries into a sophisticated algorithm is guaranteed to yield better reuse, our simple optimization algorithm mostly relies on ASR improvement through approximate batching, which is insufficient for AMD GPU models. Recorded ASR values are listed in Table 2a.

Similar trends can be observed for our evaluation on Intel GPUs (Table 2b). Since their architecture features the biggest batch size and the largest cache, the achieved ASR values with cache-based algorithms are even closer to ideal reuse rates. Additionally, as previously mentioned, batching alone does not suffice to fully explain the formation of the shading rate on Intel. Hence, our simple batch-focused algorithm is on par with Hoppe's optimization, but newer, cache-focused algorithms provide a better choice overall.

6 DISCUSSION

Although modern GPU hardware evolves at a staggering speed, the existence of a central, post-transform vertex cache is still considered to be self-evident by many. Our experiments conclusively show that this assumption does no longer match the behavior that we observe on these massively parallel devices. The most reasonable, and in fact, most likely alternative is the batch-driven decomposition of input vertex data, so that it can be efficiently handled in separate, independent processing units. By drawing a strong association between the hardware design and information

Table 2. ASR for all test scenes with optimization techniques applied. We provide recorded results for AMD RX Vega 56 (virtually identical to those by the R7 200) and Intel HD 630. Although the performance of our simple batch-based method is usually on par with more sophisticated techniques, it is commonly bested by K-Cache and Tipsify.

(a) AMD

	Hoppe	K-Cache	Tipsify	TomF	Batching
sph	**0.66**	0.67	0.68	0.77	0.72
bun	**0.68**	0.70	0.72	0.77	0.72
bud	0.73	**0.71**	0.75	0.74	0.75
dra	**0.67**	0.69	0.71	0.77	0.72
tree	2.06	2.07	2.07	2.06	**2.06**
am1	0.85	**0.74**	0.76	0.77	0.77
am2	0.81	0.68	0.68	0.69	**0.68**
as1	0.74	**0.73**	0.75	0.74	0.75
as2	1.19	**1.19**	1.20	1.19	1.20
dx1	0.77	**0.75**	0.79	0.75	0.82
dx2	0.75	**0.73**	0.76	0.74	0.74
sg1	0.73	**0.71**	0.75	0.73	0.74
sg2	0.77	**0.73**	0.77	0.75	0.76
sh1	0.88	**0.84**	0.86	0.84	0.84
sh2	0.87	**0.85**	0.88	0.86	0.86
tr1	0.83	**0.78**	0.81	0.79	0.80
tr2	0.81	**0.78**	0.81	0.79	0.79
tw1	**0.72**	0.73	0.75	0.78	0.75
tw2	1.35	**1.31**	1.33	1.32	1.32

(b) Intel

	Hoppe	K-Cache	Tipsify	TomF	Batching
sph	0.59	0.58	**0.52**	0.58	0.60
bun	0.60	0.53	**0.53**	0.57	0.58
bud	0.61	**0.55**	0.55	0.55	0.62
dra	0.60	0.53	**0.52**	0.57	0.60
tree	2.06	2.06	2.06	2.06	**2.06**
am1	0.72	**0.60**	0.60	0.65	0.69
am2	0.67	**0.48**	0.48	0.55	0.60
as1	0.62	**0.59**	0.60	0.60	0.64
as2	1.14	**1.11**	1.12	1.12	1.14
dx1	0.64	0.63	0.65	**0.62**	0.73
dx2	0.63	**0.62**	0.62	0.62	0.65
sg1	0.57	**0.54**	0.55	0.55	0.60
sg2	0.61	**0.56**	0.57	0.58	0.63
sh1	0.79	**0.74**	0.74	0.75	0.77
sh2	0.77	**0.74**	0.75	0.77	0.78
tr1	0.72	**0.68**	0.68	0.69	0.71
tr2	0.70	**0.66**	0.67	0.67	0.70
tw1	0.63	**0.57**	0.57	0.60	0.63
tw2	1.29	**1.23**	1.24	1.24	1.27

obtained through indirect measurements, we have shown that it is possible to predict this decomposition process with high accuracy. Understanding the clustering and distribution of vertex information on GPUs has several important implications for research into mesh optimization algorithms: previous work commonly uses cache-based software simulations to illustrate the merit of their work. However, during our evaluation, we have shown that the actually achieved vertex shading rates differ significantly from those reported by such simulations. Knowledge of the GPU batch functions provides us with a reliable tool for off-line simulation and evaluation of mesh optimization algorithms. Given the exact parameters of the batch function, we were able to devise a preprocessing algorithm to improve the vertex shading rate on modern GPUs beyond the capabilities of previous work. Furthermore, since hardware simulation and state prediction is oftentimes included in the optimization algorithm itself, our work opens the door for the development of new algorithms that may achieve even lower shading rates and better vertex reuse on contemporary GPU hardware. We have made our source code for reuse analysis and optimization available to the public at https://github.com/GPUPeople/vertex_batch_optimization. Lastly, challenging a concept as established as the post-transform cache and finding a more precise fit raises the question what other, long-held assumptions about the conventional graphics pipeline should be revisited. We hope our work can be a first step in the reevaluation of GPU processing as it is generally understood.

ACKNOWLEDGMENTS

This research was supported by the German Research Foundation (DFG) grant STE 2565/1-1, and the Austrian Science Fund (FWF) grant I 3007.

REFERENCES

Joshua Barczak. 2016. Vertex Cache Measurement. http://www.joshbarczak.com/blog/?p=1231 Retrieved: June 4th, 2018.

Jatin Chhugani and Subodh Kumar. 2007. Geometry Engine Optimization: Cache Friendly Compressed Representation of Geometry. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D '07)*. ACM, New York, NY, USA, 9–16. https://doi.org/10.1145/1230100.1230102

Mike M. Chow. 1997. Optimized Geometry Compression for Real-time Rendering. In *Proceedings of the 8th Conference on Visualization '97 (VIS '97)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 347–ff. http://dl.acm.org/citation.cfm?id=266989.267103

Michael Deering. 1995. Geometry Compression. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '95)*. ACM, New York, NY, USA, 13–20. https://doi.org/10.1145/218380.218391

Francine Evans, Steven Skiena, and Amitabh Varshney. 1996. Optimizing Triangle Strips for Fast Rendering. In *Proceedings of the 7th Conference on Visualization '96 (VIS '96)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 319–326. http://dl.acm.org/citation.cfm?id=244979.245626

Tom Forsyth. 2006. Linear-speed vertex cache optimisation. https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html

Fabian Giesen. 2011. A trip through the Graphics Pipeline 2011. https://fgiesen.wordpress.com/2011/07/03/a-trip-through-the-graphics-pipeline-2011-part-3/ Retrieved: June 4th, 2018.

Songfang Han and Pedro V. Sander. 2016. Triangle Reordering for Reduced Overdraw in Animated Scenes. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '16)*. ACM, New York, NY, USA, 23–27. https://doi.org/10.1145/2856400.2856408

Hugues Hoppe. 1999. Optimization of Mesh Locality for Transparent Vertex Caching. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 269–276. https://doi.org/10.1145/311535.311565

Intel Corporation 2013. *Developer's Guide for Intel® Processor Graphics For 4th Generation Intel® Core™ Processors.* Intel Corporation.

Martin Isenburg and Peter Lindstrom. 2005. Streaming meshes. In *IEEE Visualization*. 231– 238.

Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* abs/1804.06826 (2018). arXiv:1804.06826 http://arxiv.org/abs/1804.06826

Michael Kenzel, Bernhard Kerbl, Wolfgang Tatzgern, Elena Ivanchenko, Dieter Schmalstieg, and Markus Steinberger. 2018. On-the-fly Vertex Reuse for Massively-Parallel Software Geometry Processing. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2, Article 28 (Aug. 2018), 17 pages. https://doi.org/10.1145/3233303

Christoph Kubisch. 2015. *Life of a triangle – NVIDIA's logical pipeline.* Technical Report. NVIDIA Corporation. https://developer.nvidia.com/content/life-triangle-nvidias-logical-pipeline

Gang Lin and Thomas P. Y. Yu. 2006. An improved vertex caching scheme for 3D mesh rendering. *IEEE TVCG* 12, 4 (July 2006), 640–648. https://doi.org/10.1109/TVCG.2006.59

Tim Purcell. 2010. Fast Tessellated Rendering on the Fermi GF100. In *High Performance Graphics Conf., Hot 3D presentation.*

Guennadi Riguer. 2006. The Radeon X1000 Series Programming Guide.

Pedro V. Sander, Diego Nehab, and Joshua Barczak. 2007. Fast Triangle Reordering for Vertex Locality and Reduced Overdraw. *ACM Trans. Graph.* 26, 3, Article 89 (July 2007). https://doi.org/10.1145/1276377.1276489

Jeremy W. Sheaffer, David Luebke, and Kevin Skadron. 2004. A Flexible Simulation Framework for Graphics Architectures. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '04)*. ACM, New York, NY, USA, 85–94. https://doi.org/10.1145/1058129.1058142

Marc Tchiboukdjian, Vincent Danjean, and Bruno Raffin. 2008. A Fast Cache-Oblivious Mesh Layout with Theoretical Guarantees. In *International Workshop on Super Visualization (IWSV'08)*. Kos, Greece. https://hal.inria.fr/inria-00436053

Marc Tchiboukdjian, Vincent Danjean, and Bruno Raffin. 2010. Binary Mesh Partitioning for Cache-Efficient Visualization. *IEEE TVCG* 16, 5 (Sept 2010), 815–828. https://doi.org/10.1109/TVCG.2010.19

Huy T. Vo, Claudio T. Silva, Luiz F. Scheidegger, and Valerio Pascucci. 2012. Simple and Efficient Mesh Layout with Space-Filling Curves. *Journal of Graphics Tools* 16, 1 (2012), 25–39. https://doi.org/10.1080/2151237X.2012.641828

Po-Han Wang, Chia-Lin Yang, Yen-Ming Chen, and Yu-Jung Cheng. 2011. Power Gating Strategies on GPUs. *ACM Trans. Archit. Code Optim.* 8, 3, Article 13 (Oct. 2011), 25 pages. https://doi.org/10.1145/2019608.2019612

Sung-eui Yoon and Peter Lindstrom. 2007. Random-Accessible Compressed Triangle Meshes. *IEEE TVCG* 13, 6 (Nov 2007), 1536–1543. https://doi.org/10.1109/TVCG.2007.70585