On-the-fly Vertex Reuse for Massively-Parallel Software Geometry Processing

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

(a) shading rate visualization (b) game scene rendering (c) inner and outer mesh envelopes

Fig. 1. To evaluate the effectiveness and performance of our on-the-fly vertex reuse strategies, we have implemented a variety of test applications. (a) Visualization of the shading rate achieved during shading of the vertices of a triangle mesh (green vertices are shaded only once, red vertices six or more times). (b) A full rasterization pipeline rendering scene geometry captured from the video game *The Witcher 3: Wild Hunt*. (c) In general, many geometric algorithms can benefit from vertex reuse, such as the simplification envelopes algorithm shown here. *The Witcher 3: Wild Hunt screenshot courtesy of CD PROJEKT S.A.; used with permission.*

Due to its flexibility, compute mode is becoming more and more attractive as a way to implement many of the algorithms part of a state-of-the-art rendering pipeline. A key problem commonly encountered in graphics applications is streaming vertex and geometry processing. In a typical triangle mesh, the same vertex is on average referenced six times. To avoid redundant computation during rendering, a post-transform cache is traditionally employed to reuse vertex processing results. However, such a vertex cache can generally not be implemented efficiently in software and does not scale well as parallelism increases. We explore alternative strategies for reusing per-vertex results on-the-fly during massively-parallel software geometry processing. Given an input stream divided into batches, we analyze the effectiveness of sorting, hashing, and intra-thread-group communication for identifying and exploiting local reuse potential. We design and present four vertex reuse strategies tailored to modern GPU architectures. We demonstrate that, in a variety of applications, these strategies not only achieve effective reuse of vertex processing results, but can boost performance by up to 2–3× compared to a naïve approach. Curiously, our experiments also show that our batch-based approaches exhibit behavior similar to the OpenGL implementation on current graphics hardware.

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

Additional Key Words and Phrases: Vertex Processing, GPU

Authors' address: Michael Kenzel, michael.kenzel@icg.tugraz.at; Bernhard Kerbl, bernhard.kerbl@icg.tugraz.at; Wolfgang Tatzgern, wolfgang.tatzgern@student.tugraz.at; Elena Ivanchenko, elena.ivanchenko@icg.tugraz.at; Dieter Schmalstieg, schmalstieg@tugraz.at; Markus Steinberger, steinberger@icg.tugraz.at, Graz University of Technology, Institute of Computer Graphics and Vision, Inffeldgasse 16, Graz, 8010, Austria.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, https://doi.org/10.1145/3233303.

ACM Reference Format:
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 (August 2018), 17 pages. https://doi.org/10.1145/3233303

1 INTRODUCTION

Although hardware-supported, real-time rendering of 3D scenes is highly efficient, the standard rendering pipeline implemented in hardware lacks flexibility in certain aspects. With modern graphics processing units (GPU) inexorably rising in compute power, implementing (parts of) custom pipelines in compute-mode, *i.e.*, in software using CUDA, OpenCL, or compute shaders, becomes an interesting alternative. Although certain features—like rasterization—will likely always be multiple orders of magnitude faster in hardware and only accessible using OpenGL or Direct3D in 3D rendering mode, others may efficiently be realized also in software. Primitive transformations, *i.e.*, vertex shading, is one of those applications. While implementing vertex shading stages in software for execution on GPU compute units becomes more and more common, *vertex reuse*, *i.e.*, reusing the result of the vertex shader when it is referenced more than once, is usually ignored. This is in part due to vertex reuse being realized in hardware in the conventional pipeline and not exposed for custom use in compute-mode.

However, vertex reuse should not be neglected, as it can greatly reduce the number of shader invocations. A vertex in a mesh is, on average, referenced up to six times. The traditional solution to enable vertex reuse is the employment of a *post-transform cache*. The post-transform cache stores shaded vertex information, which can then be retrieved instead of computing the same information multiple times [Sheaffer et al. 2004; Wang et al. 2011]. The significance of this assumption is underlined by the wide body of research aiming at improving the ordering of vertices in meshes to yield better cache behavior. Unfortunately, there is little publicly available information on the implementation specifics used in current GPUs.

The adequacy of a central vertex cache in contemporary graphics pipelines is questionable considering recent articles [Kubisch 2015; Kubisch and Boudier 2016; Purcell 2010]. Thus, the use of such a cache in a software pipeline should also be questioned, and justifiably so: with the increasing degree of parallelism usually present in modern GPUs, the costs of a post-transform cache can be expected to rise drastically. Alternative design choices tailored towards massively parallel devices may circumvent this bottleneck while achieving similar or even better reuse characteristics. In this light, we see large potential benefits by revisiting the problem of efficient vertex reuse with an additional focus on software rendering pipelines. In search of methods capable of scaling with the massively parallel architecture of current and future GPUs, we make the following contributions:

(1) We investigate batch-based vertex uniquization as an alternative to post-transform caching for achieving reuse.
(2) Next to a naïve processing scheme, we discuss four batch-based approaches to identify unique vertices on massively parallel devices.
(3) We evaluate all approaches with respect to their theoretical and practical vertex reuse effectiveness in a variety of computer graphics applications.

2 RELATED WORK

It has been realized early on that there is significant potential for optimization by minimizing redundancy in an input stream describing mesh geometry. The pioneering work by Deering [1995], Evans et al. [1996], and Chow [1997] considered the problem from a data compression point of view. However, due to this angle of approach, these methods required input geometry to always first

be encoded according to some compression scheme which would then be decompressed during processing.

Hoppe [1999] was the first to explore the use of a k-FIFO post-transform *vertex cache* to reduce redundant vertex processing on-the-fly during rendering of triangle meshes. They furthermore presented a set of algorithms that automatically optimize the rendering sequence for a given mesh to maximize utilization of their proposed cache architecture. The downside of their approach is that it requires exact knowledge of the properties of the underlying hardware which are subject to change. However, their work inspired a long line of followup research improving upon their results [Chhugani and Kumar 2007; Lin and Yu 2006; Sander et al. 2007]. Arguably one of the most impactful works is the architecture-agnostic approach by Forsyth [2006].

A more current area of research where we encounter the problem of massively parallel vertex processing is software rendering on the modern GPU. While general pipeline architectures [Steinberger et al. 2012, 2014] focus on managing data between rendering stages, specific software rendering pipelines have been proposed for various applications [Laine and Karras 2011; Liu et al. 2010; Patney et al. 2015; Sattlecker and Steinberger 2015]. Noteworthy examples of GPU software rendering pipelines include Freepipe [Liu et al. 2010], CUDARaster [Laine and Karras 2011], and Piko [Patney et al. 2015]. They all use the compute mode of the GPU (typically on top of the CUDA [NVIDIA 2016] ecosystem) to implement rasterization and fragment shading, but lack mechanisms for vertex reuse. Freepipe simply executes the vertex shader every time an index is fetched. CUDARaster and Piko run the vertex shader in a preprocessing step on the entire vertex buffer and store the results in global memory. The need for buffering the entire intermediate output of the geometry stage leads to costly memory bandwidth requirements. Compute mode rendering is also becoming increasingly relevant in conjunction with hardware-supported rendering. For example, culling in a compute preprocessing step [Haar and Aaltonen 2015; Wihlidal 2016], can significantly improve performance. Our approach can be directly applied to such techniques.

Vertex shader result reuse is certainly considered by current GPU hardware, unfortunately, only few details have been published by the hardware vendors. While it is often assumed that modern GPU architectures are similar in design to Pomegranate [Eldridge et al. 2000], vertex reuse does not play a large role in that design. It is known, that earlier GPU generations still relied on a post transform cache [Riguer 2006], which was also features in GPU simulators of that time [Sheaffer et al. 2004]. For current NVIDIA GPUs, it has been reported that they "create batches of up to 32 triangles and 32 vertices" [Kubisch and Boudier 2016]—indicating that current GPUs apply similar techniques to our proposed approaches.

In order to avoid ambiguity in the following sections, we will employ the nomenclature for parallel execution and hardware concepts according to CUDA [NVIDIA 2016]. Hence, wavefronts of single-instruction-multiple-data (SIMD) width will be referred to as *warp*. Warp divergence indicates the case where threads follow redundant execution paths, since warps advance in lockstep. Logical groups of warps that run on the same multiprocessor share a portion of fast local *shared memory* and can easily synchronize. They will be addressed as *blocks*.

3 VERTEX REUSE STRATEGIES

A major goal of this work is a characterization of vertex reuse in a software-based, massively parallel context, and heightening the understanding of its influence on graphics workload. To this aim, we formulate the following assumptions: We only consider indexed triangles as primitives, for which the index buffer can be used to identify recurring vertices. The routine (or *shader*) for processing a vertex is invoked based on an index buffer, where groups of threads are assigned to consecutive primitives in the index buffer. In the ideal case, the vertex shader should be executed only once for each vertex that is referenced by the index buffer. To ensure high performance, shading must

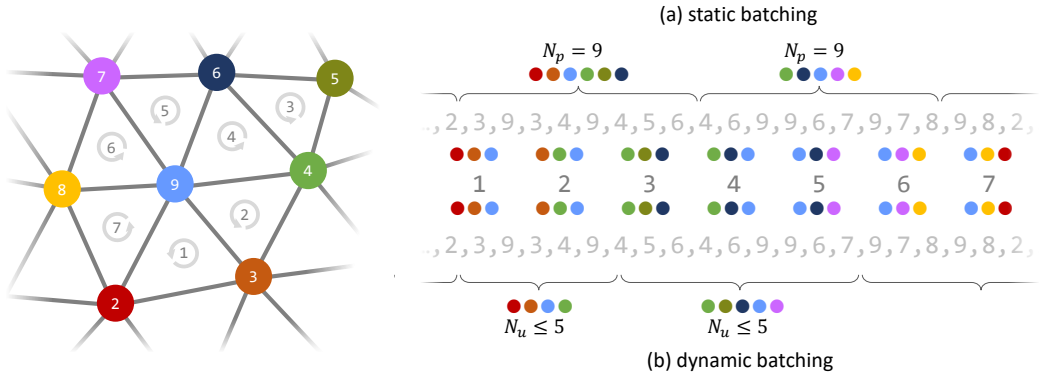


Fig. 2. Section of a mesh and its representation as indexed triangle list: On average, each vertex is referenced six times. The index buffer can be divided into (a) batches of a constant size N_p (static batching) or (b) batches of a variable size where the number of unique vertices stays below a threshold N_u (dynamic batching).

happen in parallel, without any need for expensive synchronization or communication across GPU multiprocessors. In streaming pipelines, preprocessing of the vertex or index buffer should be avoided, as the introduced read-then-write memory bandwidth can reduce overall performance.

3.1 Post-transform cache

Given the above considerations for geometry processing in a streaming pipeline, a global persistent post-transform cache is a possible choice to reduce the number of vertex shader invocations. However, such a cache is difficult to implement efficiently if it is required to work across all multiprocessors on the GPU. Furthermore, even in a single multiprocessor, the high level of data concurrency may defeat the purpose of caching. The reuse of vertex information can occur almost instantly, if neighboring triangles are referenced in quick succession in the index buffer (*e.g.*, triangle strip layout). Consequently, an advancing wave front of threads may process the same vertex multiple times in parallel, and new cache entries become available too late to be of use. In a software-only implementation, caching additionally suffers from high latency when using conventional memory rather than dedicated cache hardware.

3.2 Batch-based vertex reuse

To avoid the issues raised by the use of a central cache, we propose the concept of *batch-based* vertex processing, which naturally lends itself to execution on massively parallel architectures. We define a batch as a bounded region in the index buffer, *i.e.*, a set of triangles, which is assigned to a single warp or block for processing. The block is responsible for executing the shader once for each referenced vertex within its batch and assembles the output triangles. Each block must analyze its batch, assign vertices uniquely to threads for shader invocation and finally distribute shading results for assembling the output triangles. This implies that duplicate indices in the batch need to be identified before executing the vertex shader.

An obvious challenge in the parallel generation of this many-to-one mapping is that the input-to-output ratio is not known in advance. Ideally, we would like to choose a batch such that the number of unique vertices equals the block size, and each thread runs exactly one instance of the vertex shader. If that is not the case, under-utilization will arise, as threads receiving no vertex to process will idle. With larger batch sizes, a larger number of duplicate indices can be detected, at the cost

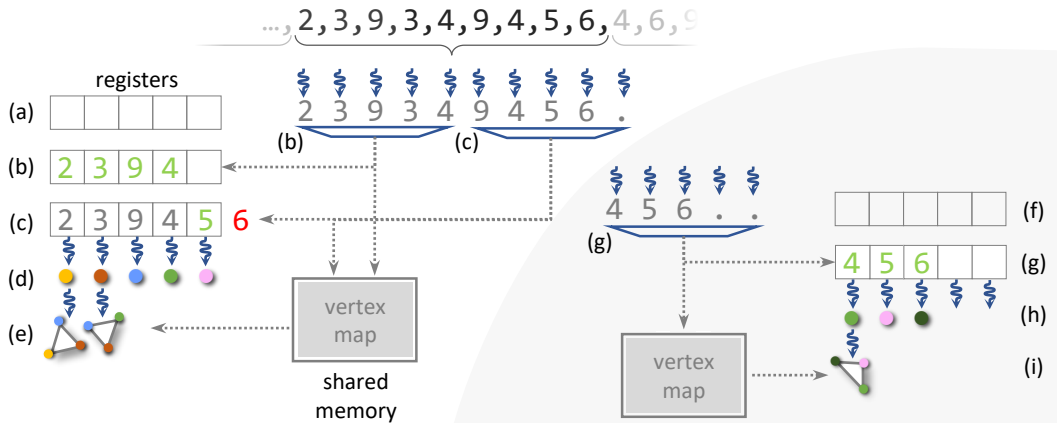


Fig. 3. Statically-batched warp voting uses all threads in a warp (5 in this example) to load indices. (b) We exploit warp voting and shuffle instructions to unify the indices and store the result in registers. (c) This process is repeated until all indices have been consumed, or all threads have acquired a unique index for processing in the vertex shader. As primitive size must be considered (e), early shading results might be discarded (*e.g.* for index 5 above). This entire process is repeated until the batch is consumed (gray, f–i).

of requiring multiple rounds of shader invocations to finish a whole batch. These considerations lead to the proposal of two strategies outlined in Figure 2: *static batching* and *dynamic batching*.

3.3 Static batching

For static batching, each block simply fetches a fixed number of indices from the input buffer to process. As a guideline for efficient processing, we use a common multiple of the block size and the primitive size as batch size, *e.g.*, for triangles and block size 32, we could use any multiple of $3 \cdot 32 = 96$. Since the batch size is fixed, static batching requires no preprocessing of the index buffer and can be applied directly to the input of a streaming pipeline.

Statically-batched naïve. As a baseline, we implement a naïve strategy that does not attempt any vertex reuse. Instead, every thread is directly assigned to a primitive, and invokes the vertex shader for all its indices. As blocks always fetch the same number of indices, the static batch size is implicitly given. Notice that, while this strategy leads to duplicate vertex shader execution, it avoids all communication overhead. Thus, for very simple vertex shaders, this naïve approach may in fact show very good performance.

Statically-batched warp voting. For this strategy, we aim to fill up warps with triangles so that every thread receives a unique vertex to work on, as detailed in Algorithm 1. Figure 3 provides an example: (a–b) Every thread first loads an index from the buffer and subsequently publishes it via register shuffle instructions to all other threads in the warp (line 10). Each thread then informs its peers via warp voting whether a duplicate index has been found (line 11). We track the number of unique indices observed so far and assign each new index to the available thread with the lowest ID (line 14). We also maintain an inverse lookup-table in shared memory for fast reassembly after shading (line 19). (c) We keep fetching indices until either all threads were assigned a unique vertex, or the batch boundary is hit. (d,e) Next, all identified unique vertices are shaded and output assembly is carried out, relying on the data from shared memory and again using efficient intra warp communication (line 26–31).

Algorithm 1: Statically-batched warp voting.

```
1  shared map[ ]                                              // map array in shared memory
2  cStart ← BatchBegin
3  while cStart < BatchEnd do
4      fill ← 0, done ← 0, my_id ← −1, offset ← cStart
5      while offset < BatchEnd and fill < WarpSize do
6          incoming ← −1, outgoing ← −1
7          if offset + laneId < BatchEnd then
8              incoming ← indexBuffer[offset + laneId]        // laneId: thread's id in warp
9          for i ∈ WarpSize do
10             curr ← shfl (incoming, i)                      // shfl: access ith thread incoming variable
11             match ← ballot (curr = my_id)                  // ballot: warp-wide bitmask (bit set if true)
12             if match = 0 then
13                 if fill = laneId then
14                     my_id ← curr
15                 match ← BitShift (1, fill)
16                 fill ← fill + 1
17             if i = laneId then
18                 outgoing ← match
19         map[done + laneId] ← ffs (outgoing)-1              // ffs(·) − 1: index of the first set bit
20         firstmask ← ballot (outgoing = 0 or incoming = −1)
21         additional ← min (WarpSize, ffs (firstmask))
22         done ← done + additional
23         offset ← offset + WarpSize
24     triangles ← ⌊done/3⌋
25     if laneId < fill then
26         v ← shade (vertexBuffer[my_id])                    // shade: vertex shader call
27     v0 ← shfl (v, map[3 · laneId])
28     v1 ← shfl (v, map[3 · laneId + 1])
29     v2 ← shfl (v, map[3 · laneId + 2])
30     if laneId < triangles then
31         output (v0,v1,v2)                                  // output: forward triangle to next stage
32     cStart ← 3 · triangles
```

(f–i) This entire process is repeated, until all indices in the batch have been processed. Note that starting a new iteration can lead to duplicate shader invocation inside a batch, since shading results are not carried over from the previous iteration; (g) reshades index 4 and 5. Due to the static batch size and the iterative assignment within a warp, a varying number of triangles is generated during each iteration; (e) produces two triangles, (i) only a single one. Furthermore, during the last iteration potentially only a fraction of threads may perform vertex shading; (g) only shades three vertices.

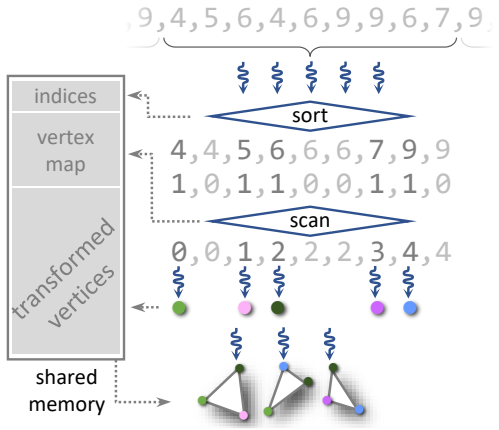


Fig. 4. Dynamically-batched sorting computes a prefix sum over sorted indices to identify unique vertices.

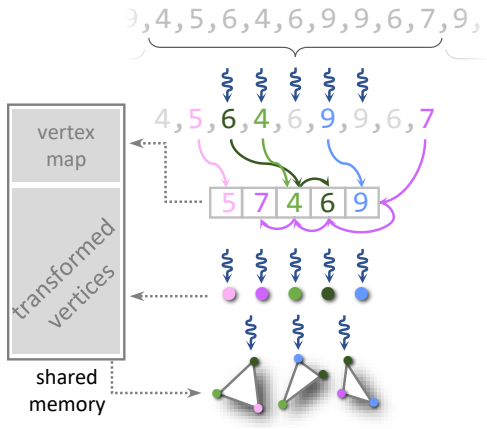


Fig. 5. Dynamically-batched hashing uses a hash map that holds one entry per thread to remove duplicates.

3.4 Dynamic batching

We assess the potential for optimizing vertex processing by allowing for a fast, low-impact preprocessing step to retrieve analytical data from the submitted index buffer. Specifically, we investigate the performance of several dynamic batching strategies, which rely on a load-time analysis of the input to derive optimal batch sizes. This routine splits the buffer into batches of variable length, with the goal of maximizing thread occupancy at runtime for the loading and processing of vertices.

To this end, we define N to be a multiple of the block size and scan the index buffer front to back, counting unique indices until we reach N, or a maximum allowed number of batch primitives is reached. When either of these conditions is met, we start a new batch and continue scanning the index buffer until all indices have been assigned to batches. We store the batch starting positions in an auxiliary buffer, which allows us to feed a close-to-ideal amount of data to each block. Note that we do not require the buffer to forward information about the unique vertices, and leave their identification to be conducted at runtime. Hence, no information other than the splitting of the index buffer into optimally processable portions is output at this point. Therefore, we can abstract our preprocessing procedure to an elaborate work scheduling routine, that could very well be realized by an initial streaming step or dedicated hardware.

Limiting the number of unique indices allows dynamically-batched approaches to handle entire batches at once. While in the static case we had to support breaking apart the input batch under resource overflow—a feature that is only trivially supported by some sort of serialization (*cf.* the sequential assign step in warp voting). Ruling out this circumstance allows dynamically-batched approaches to employ completely parallel strategies, such as *sorting*, *hashing*, or *parallel hashing*.

Dynamically-batched sorting. One way to assign unique vertices to threads is to use parallel sorting, as outlined in Algorithm 2 and Figure 4. We load and sort a full batch of indices in shared memory (line 2–5). Looking at pairs of sorted indices, we identify unique vertices and mark each first occurrence (line 7). Run a prefix sum over this data, assign unique vertices to threads (line 8–11) and run the vertex shader (line 13). Using the original position in the batch which we carried along during sorting and the mapping delivered from the prefix sum, we construct an inverted lookup-table for assembling the output triangle (line 15). Communication uses shared memory.

Algorithm 2: Dynamically-batched sorting.

```
1  shared ids[ ], linIds[ ], map[ ], marks[ ], uniqueIds[ ], v[ ]
2  for i ∈ size(Batch) do in parallel
3      ids[i] ← indexBuffer[BatchBegin + i]
4      linIds[i] ← i
5  RadixSort (ids, linIds)
6  for i ∈ size(Batch) do in parallel
7      marks[i] ← 1 if ids[i] ≠ ids[i + 1] else 0
8  numVertices ←PrefixSum (marks)
9  for i ∈ size(Batch) do in parallel
10     map[linIds[i]] ← marks[i]
11     uniqueIds[marks[i]] ← ids[i]
12 for j ∈ numVertices do in parallel
13     v[j] ← shade (vertexBuffer[uniqueIds[j]])
14 for i ∈ size(Batch)/3 do in parallel
15     output (v[map[3i], v[map[3i + 1], v[map[3i + 2]])
```

Algorithm 3: Dynamically-batched hashing.

```
1  shared hashtable[ ], map[ ], v[ ]
2  for i ∈ BatchSize do in parallel
3      hashtable[i] ← -1
4  for i ∈ size(Batch) do in parallel
5      id ← indexBuffer[BatchBegin + i]
6      p ← hash (id)
7      while not inserted do
8          prev ← atomicCAS (hashtable[i], −1, id )
9          if prev − 1 or prev = id then
10             loc ← p
11         else
12             p ← probing (p)
13     map[i] = loc;
14 for j ∈ BatchSize do in parallel
15     if hashtable[j] ≠ −1 then
16         v[j] ← shade (vertexBuffer[hashtable[j]])
17 for i ∈ size(Batch)/3 do in parallel
18     output (v[map[3i], v[map[3i + 1], v[map[3i + 2]])
```

Dynamically-batched hashing. In this strategy, we employ a hash map in shared memory to remove duplicate vertex indices, as outlined in Algorithm 3 and Figure 5. We choose the size of the hash map to match the block size. Using multiplicative hashing on the index (line 6) and an atomic compare-and-swap operation, each thread inserts its index into the hash map (line 8). If

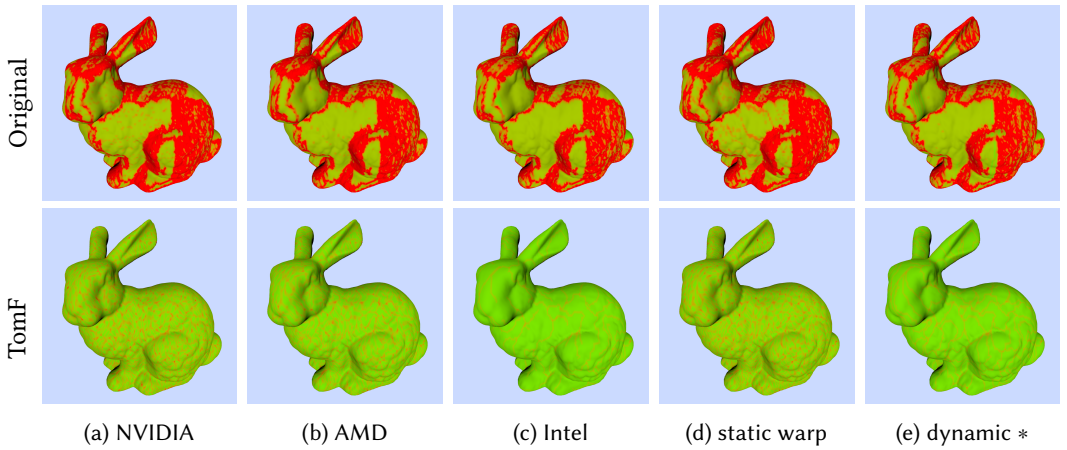


Fig. 6. Vertex reuse visualization for the Stanford Bunny model: green indicates a single shader invocation, red indicates six shader calls. Compared to the original input, preprocessing the model with TomF enables better overall potential for vertex reuse for both OpenGL (on NVIDIA GTX 1080Ti, AMD RX Vega 56, Intel HD 630) and our software techniques. Dynamic batching shows higher reuse than statically-batched warp voting due to its larger batch size (1023 vs 96). Interestingly, NVIDIA shows similar reuse to our warp voting and dynamic batching seems similar to Intel; AMD appears to be somewhere between the two.

the operation succeeds or the index is already present at this location the index assignment is completed (line 10). If the position is used by another index, we perform linear probing (line 12). Upon entering an index into the hash map, the loading thread records the value of the hash function, to identify the required vertex after shading (line 13). Ideally, the hash map is fully filled after loading a batch due to the size restrictions applied in our preprocessing step. Consequently, filling the hash map allows us to uniquely assign vertices to threads. Subsequently, we perform shading and triangle output again using shared memory for communication (line 16–18).

Dynamically-batched parallel hashing. One issue with the hashing approach above is that a fully occupied hash map will likely lead to excessive probing. This may lead to pathological warp divergence, as a single thread repeatedly tries to find the last free entry, and the remaining peers in the warp have to join in the effort. As a remedy, we propose to perform hashing as a two-tiered approach. First, every thread executes up to a fixed number of linear probing attempts. Second, all threads within a warp collaborate to find available spots until all indices have been inserted. This fast-path/slow-path strategy effectively repurposes otherwise idle threads to speed up the search for free spots. Coordination within a warp is realized through shuffle and warp voting.

4 EVALUATION

For performance evaluation, we use a set of commonly processed models, as well as content captured from five recent video games and an NVIDIA technical demo for which we merge all draw calls into a single mesh: 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). A representative rendering from our 19 different scenes is shown in Figure 1b. As measure of vertex reuse, we report the average shading rate (ASR), which is identical to the average cache miss ratio, commonly measured for cash-based approaches: sum of vertex shader invocations divided by the number of triangles. To show the usefulness of

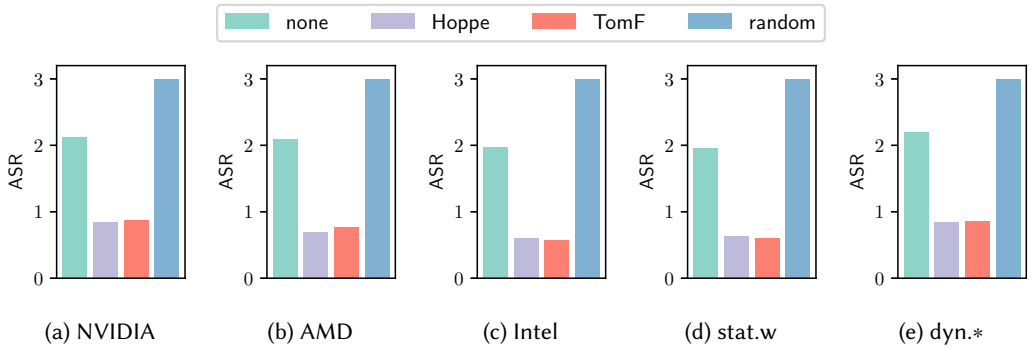


Fig. 7. Preprocessing the Stanford Bunny for vertex locality significantly improves its reuse potential. Independent of the approach, the original mesh results in about two vertex shader invocations per triangle. A random order achieves zero reuse and thus three shader invocation. Preprocessing with TomF [Forsyth 2006] or Hoppe [Hoppe 1999] reduces the number of shader calls to similar levels; which algorithm works better varies slightly among the reuse techniques.

our proposed approaches, we look at the ASR achievable by a cache-based approach and compare to vertex reuse rates achieved using OpenGL on various hardware. Furthermore, we investigate the performance of our techniques in a software streaming rendering pipeline and compare to a non-streaming, multi-kernel setup.

Obviously, the order in which vertices are referenced in input models has an influence on the ASR of different techniques. Popular mesh processing algorithms have been presented previously, with the aim of reordering indices in a given mesh to increase vertex locality. As shown in Figures 6 and 7, applying such algorithms to popular models can remove unusual discontinuities, and significantly improve reuse potential in the hardware rendering pipeline and our techniques. In order to generate a fair ordering and enable vertex reuse even in unstructured models, we preprocess all meshes with the optimization algorithm by Forsyth [2006].

4.1 Caching vs Batching and OpenGL

We first determine the ideal reuse rate as the ratio of duplicate vertex indices over the total length of the index buffer. Theoretically, a very large, global post-transform cache with instant reusability could yield the reported ideal figures. Unfortunately, such a global vertex cache does not seem practical for modern GPUs. Storing and retrieving data from a global device-wide cache would require significant cache sizes and placement in a further-away layer in the memory hierarchy—with typical latencies about an order of magnitude higher than caches on the multiprocessor. Furthermore, as cache entries can only be generated after vertex shader execution, all threads that concurrently receive the same non-cached index to execute, will not be captured by the cache, precluding an immense vertex reuse potential of being utilized.

A more realistic cache-based approach, is using per-multiprocessor caches. However, such a design only addresses the latency issue, but still faces the parallel execution problem, simply on another scale. To rule out such a design, we simulated variously sized per-multiprocessor least-recently-used (LRU) caches in a first experiment. For statically-batched warp voting we use a batch size of 96, which fits the warp size of 32 on the tested NVIDIA GPUs. For all dynamic batching approaches, we use a maximum batch size of 1023 indices and 256 unique vertices, and assign 256 threads to each batch, resulting in equal ASR values. To place the achievable vertex reuse of our approaches in comparison to hardware pipeline implementations, we employ a simple atomically

Table 1. Scene statistics: vertices, triangles, and average shading rate (ASR; lower is better, 3.0 is worst) in an ideal case, for a per-multiprocessor cache, OpenGL on different hardware (NVIDIA GTX 1080Ti, AMD RX Vega 56, Intel HD 630), statically-batched warp voting, and achieved by all dynamic batching approach. The cache experiments assume 1024 vertices being shaded in parallel and the cache size is given in elements.

				Parallel Cache			OpenGL			Ours	
	vert	tris	ideal	1024	2048	4096	NVIDIA	AMD	Intel	stat.w	dyn.*
bunny	34k	70k	0.504	2.832	2.820	2.799	0.879	0.770	0.571	0.858	0.603
sphere	40k	82k	0.501	2.811	2.805	2.793	0.884	0.772	0.584	0.864	0.615
tree	492k	239k	2.058	2.997	2.997	2.997	2.078	2.061	2.059	2.058	2.061
buddha	544k	1.1M	0.501	2.850	2.856	2.853	0.991	0.745	0.551	0.825	0.588
dragon	3.6M	7.2M	0.501	2.823	2.823	2.823	1.163	0.765	0.568	0.861	0.615
am02	3k	6k	0.597	2.874	2.829	2.775	0.874	0.771	0.652	0.873	0.663
am03	2k	4k	0.483	2.730	2.676	2.676	0.806	0.694	0.555	0.795	0.579
as01	108k	183k	0.591	2.898	2.895	2.895	0.860	0.743	0.603	0.843	0.636
as04	598k	538k	1.113	2.949	2.946	2.946	1.256	1.186	1.120	1.275	1.140
dx29	25k	42k	0.612	2.898	2.895	2.889	0.855	0.751	0.621	0.843	0.654
dx33	37k	60k	0.615	2.916	2.913	2.913	0.847	0.738	0.618	0.846	0.648
sg14	135k	254k	0.534	2.871	2.868	2.868	0.841	0.728	0.547	0.822	0.585
sg16	38k	69k	0.561	2.859	2.856	2.856	0.855	0.748	0.575	0.840	0.612
sh11	812k	1.1M	0.738	2.925	2.922	2.922	0.975	0.836	0.747	0.921	0.768
sh21	521k	701k	0.747	2.913	2.913	2.913	0.954	0.861	0.767	0.957	0.789
tr04	191k	283k	0.675	2.901	2.898	2.898	0.889	0.791	0.687	0.876	0.711
tr09	78k	118k	0.660	2.907	2.907	2.907	0.890	0.787	0.672	0.885	0.693
tw03	268k	487k	0.552	2.847	2.844	2.841	0.887	0.783	0.596	0.873	0.639
tw30	695k	565k	1.233	2.940	2.940	2.940	1.390	1.320	1.243	1.404	1.263

operated invocation counter in the vertex shader. The measured ASR for this instrumentation as well as all other techniques together with scene statistics are listed in Table 1.

As expected, parallel cache-based approaches, even in a distributed setup hardly reduce vertex shader invocations compared to a naïve approach (ASR 3.0). In contrast, our approaches are highly effective for all kinds of scenes. As dynamic batching works on larger batches, it always achieves a lower ASR than static batching. On average, statically-batched warp voting achieves an ASR which is 0.3 worse than ideal and dynamic batching is off by only 0.1, which is equivalent to one more vertex shader invocation every three and ten triangles, respectively. Interestingly, the hardware-based reuse approaches achieve similar results to our approaches. The approach employed on the NVIDIA GTX 1080Ti achieves a slightly worse ASR than statically-batched warp voting. Our dynamic batching approaches achieve a slightly better ASR than observed on the AMD RX Vega 65 and slightly worse results than the Intel HD 630.

4.2 Real-time Rendering

The major motivation and use case for vertex reuse is the geometry processing stage of a real-time 3D rendering pipeline. To test our batch-based reuse techniques, we have implemented a configurable geometry stage in CUDA, that can be included into a streaming pipeline design. The geometry processing stage is simply given an input stream of indices and a vertex buffer. Based on the respective batching approach, indices are fetched from the index buffer and vertex reuse is evaluated. As a final step, one thread per triangle is used to write the output primitive with its

vertices into a queue. This output queue could—when integrated into a full streaming pipeline—be consumed by the next stage in the rendering pipeline. For a traditional real-time rendering pipeline, this queue would form the natural point for a *sort-middle* approach [Molnar et al. 1994].

The runtime results for the vertex processing for selected tested techniques on an NVIDIA GTX 1080Ti are shown in Table 2; the full data set can be found in the supplemental material. To simulate different vertex shader loads, we used a simple matrix multiplication (simple), a load of 256, 512 and 1024 fused-multiply add (FMA) instructions. We also include a non-streaming, multi-staged processing implementation for reference. With this approach, all vertices in the vertex buffer are processed only once by separate kernel. The output vertex data can then be directly loaded from global memory in a separate kernel for assembling the output primitives. This approach is employed, *e.g.*, by Laine and Karras [2011] for rasterization of 3D scenes. Since vertices need to be shaded exactly once, this technique can achieve ideal reuse, but only at the cost of sacrificing the advantages of a streaming architecture. For instance, the entire intermediate data needs to go through slow global memory. Although our test only uses five output attributes per vertex and thus generates only little intermediate data, our vertex reuse techniques can even outperform multi-staged geometry processing on several accounts.

As can be seen, for a very simple vertex shader, the naïve, no-reuse approach is the fastest, as it has no communication overhead. However, warp voting and sorting are on average only about 1.5× slower, and both hashing approaches are about 2.0× behind. The results indicate that among the techniques capable of vertex reuse, warp voting has the lowest overhead. As the vertex shader load increases, the naïve approach quickly falls behind, showing that the proposed approaches can efficiently detect vertex reuse. For a 256 FMA load, warp-voting achieves the best performance, followed by parallel-hashing and hashing. For this load, the lower overhead of warp-voting still outweighs its lower vertex reuse rate. However, for larger loads the two hashing approaches catch up, and performance is overall tied between our three techniques, while sorting eventually trails behind. We attribute the high performance of both hashing approaches and their marginal difference to the efficient implementation of shared memory atomics on recent GPUs.

To assess the performance of the proposed approach across multiple GPU generations, we also tested an NVIDIA GTX 780Ti, 980Ti and report the relative execution time compared to naïve, averaged over the entire test body in Figure 8. As can be seen, there is a significant difference across GPU generations, which is mostly due to more efficient shared memory operations. While for a simple shader, all vertex reuse approaches reduce performance in comparison to naïve, the more complex shaders again benefit greatly from reuse. Although statically-batched warp voting again slightly loses ground in comparison to the other approaches on the GTX 1080Ti in the complex case, it outperforms the other approaches on average over all GPUs. Additionally, statically-batched warp voting does not require analysis of the index buffer and thus can be used in a full streaming approach.

5 SOFTWARE APPLICATIONS

In addition to rendering, we also evaluate our techniques for their potential in context of general, software-based processing tasks that allow for reuse. Specifically, we consider them for mesh subdivision and morphological transformation for inner and outer envelopes on 2-manifold models. Furthermore, we run a random walk simulation based on probabilistic input parameters. All applications were implemented in CUDA and executed on an NVIDIA GTX 1080Ti.

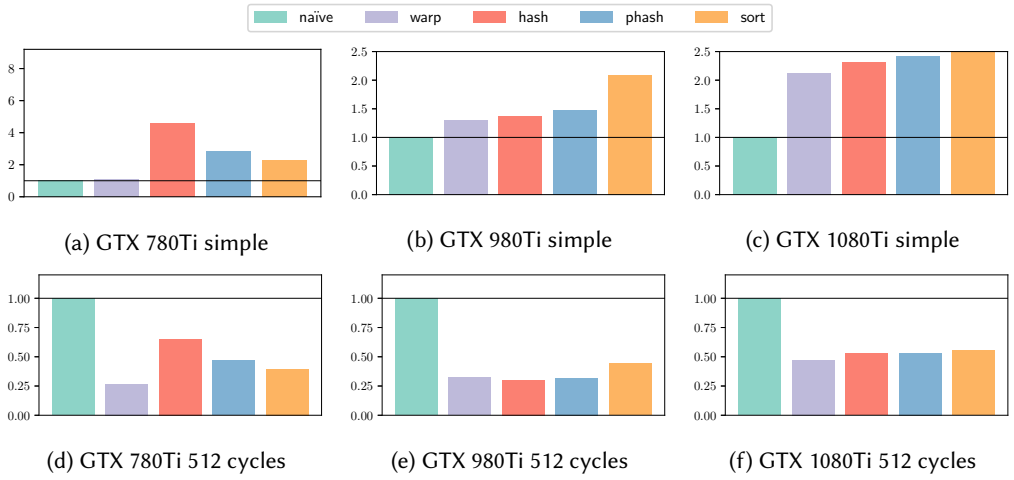
5.1 Mesh Subdivision

Subdivision of meshes can be achieved by adding primitives to a mesh. The newly introduced primitives are determined by analyzing their topological neighborhood. An easily parallelizable

Table 2. Processing times achieved with different vertex reuse techniques for rendering of geometry on a GTX 1080Ti. We also include a non-streaming, multi-kernel technique (*multi*) for comparison.

		naïve	warp	hash	phash	sort	*multi*
simple	sphere	**0.07**	0.13	0.16	0.13	0.20	*0.10*
	tree	**0.34**	0.40	0.42	0.41	0.51	*0.38*
	dragon	**4.59**	10.38	12.89	11.00	6.36	*6.24*
	buddha	**0.81**	1.69	2.05	1.82	1.21	*1.02*
	as01	**0.14**	0.24	0.27	0.23	0.31	*0.19*
	dx33	**0.05**	0.11	0.12	0.10	0.13	*0.08*
	sg14	**0.19**	0.34	0.46	0.39	0.41	*0.26*
	sh11	**0.83**	1.39	1.47	1.32	1.18	*1.09*
	tr04	**0.21**	0.36	0.40	0.35	0.34	*0.30*
	tw03	**0.35**	0.61	0.74	0.65	0.61	*0.47*
512 cycles	sphere	0.39	**0.17**	0.19	0.18	0.26	*0.13*
	tree	1.10	**0.53**	0.62	0.55	0.83	*0.68*
	dragon	25.22	**6.38**	11.2	8.48	9.57	*7.82*
	buddha	4.55	**1.13**	1.78	1.64	1.94	*1.36*
	as01	0.84	**0.27**	0.32	0.29	0.41	*0.27*
	dx33	0.29	**0.14**	0.16	0.15	0.16	*0.11*
	sg14	1.13	**0.35**	0.51	0.44	0.53	*0.38*
	sh11	4.45	**1.27**	1.4	1.32	1.89	*1.57*
	tr04	1.27	0.38	0.44	**0.37**	0.56	*0.44*
	tw03	2.10	**0.56**	0.79	0.74	0.98	*0.65*

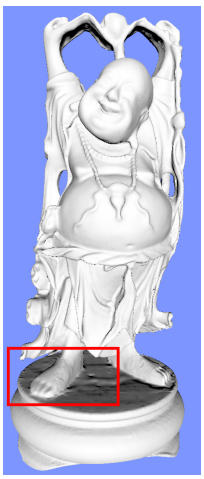
		naïve	warp	hash	phash	sort	*multi*
256 cycles		0.22	**0.13**	0.16	0.15	0.22	*0.11*
		0.66	**0.33**	0.47	0.42	0.66	*0.49*
		14.71	**5.38**	9.27	8.42	8.19	*6.62*
		2.57	**0.93**	1.60	1.59	1.69	*1.14*
		0.47	**0.19**	0.30	0.26	0.37	*0.23*
		0.17	**0.11**	0.14	0.12	0.15	*0.09*
		0.64	**0.26**	0.49	0.41	0.51	*0.31*
		2.57	**0.93**	1.39	1.28	1.56	*1.24*
		0.72	**0.27**	0.40	0.33	0.54	*0.36*
		1.18	**0.43**	0.69	0.66	0.87	*0.54*
1024 cycles		0.71	0.25	0.24	**0.23**	0.31	*0.16*
		2.02	0.98	1.00	**0.91**	1.25	*1.08*
		46.88	10.90	10.91	**10.29**	12.45	*10.24*
		8.09	**1.78**	1.98	2.01	2.34	*1.80*
		1.56	0.43	0.42	**0.37**	0.49	*0.36*
		0.53	0.21	0.22	**0.20**	0.21	*0.14*
		2.09	**0.56**	0.63	0.57	0.64	*0.51*
		7.43	2.14	1.98	**1.86**	2.53	*2.16*
		2.25	0.65	0.58	**0.54**	0.73	*0.62*
		3.75	0.95	0.9	**0.89**	1.17	*0.87*



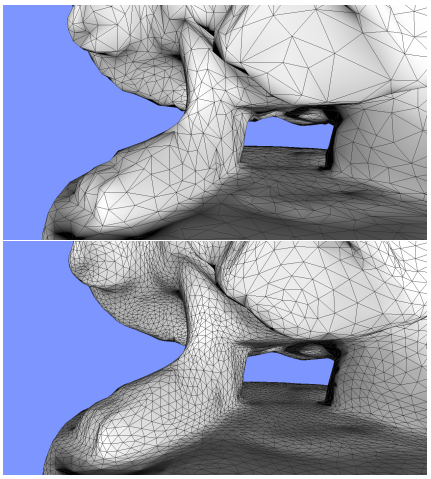
(a) GTX 780Ti simple (b) GTX 980Ti simple (c) GTX 1080Ti simple

(d) GTX 780Ti 512 cycles (e) GTX 980Ti 512 cycles (f) GTX 1080Ti 512 cycles

Fig. 8. Reported runtimes of the vertex processing stage in our software renderer, averaged over the entire test body (19 original scenes) and plotted relative to naïve. Different GPU architectures behave quite diversely: note the poor performance of hash, compared to our optimization p.hash on older architectures (GTX 780Ti), which is mostly due to the performance of shared memory atomics. Overall, statically-batched warp voting seems to be the most reliable approach.



(a) Buddha statue



(b) Wireframe close-up

	bunny	sphere	buddha
naïve	0.70	0.79	11.1
warp	0.58	0.68	8.5
hash	0.58	0.68	**8.15**
p.hash	0.59	0.69	8.17
sort	**0.57**	**0.64**	8.61

(c) Runtimes in ms

Fig. 9. Running Loop subdivision on a simplified buddha (a) with vertex streaming, creates a smoother, subdivided output (b). Performance gains can be observed across all models applying our approach (c).

subdivision algorithm has been presented by Loop [1987]. Loop subdivision produces a piecewise linear approximation of smooth surfaces based on B-spline and multivariate spline theory. For each edge and vertex, vertices are added in each subdivision iteration. The position of new vertices is computed from a convex combination of the adjacent primitives. Starting the processing from the view-point of output primitives, allows to consider vertex reuse for the output mesh, which mostly results in reducing memory accesses to the input. Figure 9 shows results for subdividing a simplified version of the original Happy Buddha model. We ran one iteration of the Loop subdivision with different vertex reuse strategies on the `bunny`, `sphere` and `buddha` models.

We evaluated a wide variety of different parameters for batch and block size, and chose those producing the best results for our final consideration. For `naïve` and `warp`, a batch size of 96 was used. For both `hash` and `p.hash`, we chose batches containing up to 192 indices and 64 unique indices/threads per block. For `sort`, best results were achieved at a batch size of 768, with a block size of 256 unique indices/threads. The Loop subdivision algorithm is arguably quite simple, and hence the cost of re-shading vertices comparably inexpensive. However, the reduction in runtime with our vertex reuse techniques can still be as high as 26%. Without exception, all vertex reuse techniques outperform the naïve approach for the tested scenes (see Figure 9c).

5.2 Simplification Envelopes

The inner/outer envelopes of a mesh are defined to occupy a strict spatial sub-/superset of the input. Resulting meshes can be used, *e.g.*, as input to a variety of simplification algorithms, manipulation of subdivision or for conservative intersection/collision testing with tolerance [Cohen et al. 1996; Zhou et al. 2007]. An envelope is obtained by moving vertices along their vertex normal towards the *inside* or the *outside* of the model. Provided that the original mesh does not contain self-intersections, an inner or outer envelope must also retain this property. Hence, before moving each vertex, we need to determine a safe distance ϵ to ensure that no intersections will occur as a result of its transformation. We have implemented the analytical approach presented by Cohen et al. [1996] in a streaming rendering scenario, where our vertex reuse strategies can be applied to the processed

Table 3. Runtimes for parallel envelope creation, given in ms. Due to the particularly high shader cost, models with good vertex reuse (sphere) can achieve 3× the performance obtained with naïve streaming alternatives.

	naïve	warp	hash	p.hash	sort
bunny	71.73	29.41	22.21	21.74	**21.85**
sphere	15.08	5.55	**4.02**	4.03	4.03
buddha	5021.80	2112.45	1595.31	1516.15	**1509.76**

triangles. Potential intersections are identified and resolved efficiently by providing an octree representation of the scene as auxiliary input. Inner and outer envelopes for the bunny model are shown in Figure 1c for a target ϵ equal to 2% of the mesh's bounding box diagonal.

We again report results with the best configuration found for each technique. As with subdivision, batch/block sizes for `naïve` and `warp` were chosen as 96/32 and 96/64, respectively. For all dynamic methods, we use a block size of 128, with a batch size of 576 for `hash`, and 768 for both `p.hash` and `sort`. The envelope creation routine is comparably complex, and the incurred cost for each "shaded" vertex in the creation of envelopes is high: computing an intersection-free offset for a vertex to move by requires traversing the octree, which entail significant global memory. Similarly to our experiments for rendering with high shader loads, a speed-up of more than 3× can be achieved over naïve streaming. Table 3 lists reported runtimes in milliseconds for processing 2-manifold models.

5.3 Parallel Random Walk

A random walk [Pearson 1905] describes a stochastic or random process, where a path is chosen on top of a graph structure or given domain, based on successive randomized steps. Random walks are used, *e.g.*, to simulate the paths of molecules traveling through liquids, the random search path of animals, or messages traversing through a social network.

To evaluate whether on-the-fly reuse computations can increase the performance of such random processes, we implemented a parallel walk on a discrete domain that follows a Levy flight [Kleinberg 2000]. We use a grid size of 256×256 and place 300 000 agents on this grid. To simulate their activity, we overlay multiple Gaussian functions on this domain. The likelihood for agents to move a certain distance is then computed based on the activity input to the Fokker–Planck equation. To evaluate the movements, we run through all potential moves with a maximum distance of 16 and keep only those 8 with the highest likelihood. Then, every agent draws a random number to choose one of the stored options, whereas each is chosen with a probability proportional to their relative likelihood.

Reuse can be implemented in this scenario as follows. We encode the current agent location as a combined integer, using half of the bits for each dimension, yielding a single 32-bit word. This number serves as a virtual "index" for the reuse computations, combining agents that are currently placed on the same grid location. Given that the movement probability only depends on the current position, all agents with combined "indices" will see identical movement likelihoods, which can be computed only once. The final step, which involves drawing a random number and choosing the most likely move, has to be carried out separately.

Initializing all 300 000 agents randomly and running 10 simulation steps on the 256×256 grid showed that our reuse strategies can significantly increase the performance of the parallel random walk. Naïve, warp, hash, p.hash and sort took 0.30 ms, 0.10 ms, 0.09 ms, 0.13 ms and 0.10 ms for one time step, respectively. The batch sizes that achieved the best performance were large (1536 for dynamic and 576 for static batching). At first glance, the great performance of reuse is not surprising, as the likelihood computations are rather time complex, and a high benefit can be expected for expensive vertex shaders. However, note that the agents are also likely to significantly diverge

throughout the random walk. A further analysis revealed that a small amount of reuse already entails a significant performance gain, as the large batch sizes can still reduce the computations.

6 CONCLUSION

Ever since its introduction by Hoppe [1999], a vertex cache has been the de facto standard approach to avoid redundant vertex shading. However, caching seems to be less applicable to modern, massively parallel devices. We have presented four inherently parallel, batch-based approaches, providing a suitable alternative to a conventional vertex cache. Our methods are straightforward to implement in software, and operate directly on an indexed triangle mesh representation, with little to no preprocessing required. Especially for complex shading routines, we showed that batching can achieve high reuse and increase performance by up to 3× over non-reuse approaches. We have evaluated both static and dynamic batching methods on a variety of applications and test cases. Due to its use of fast, warp-level communication, our static warp-voting technique is well-suited for basic shading tasks, while a dynamic, hash-based batching approach usually performs best with shaders of high complexity. Considering that vertex shaders often exhibit low-to-medium complexity and the fact that it does not require a preprocessing step, warp-voting appears to be the recommended choice for streaming pipelines written in a compute language.

Our results are obtained from experiments and test applications in CUDA, but similar approaches should also lend themselves to implementation in hardware. As vertex reuse needs to interface with vertex shading, batch sizes and efficiency considerations for warp-based execution should also be transferable into hardware design. Furthermore, vertex reuse techniques such as the ones presented here are potentially applicable to more general mesh processing and parallel graph traversal problems, where node dependencies require a similar treatment. Building on the results of this work, we were able to implement a complete streaming software graphics pipeline capable of achieving real-time rendering performance on a modern GPU [Kenzel et al. 2018]. A more detailed investigation into the vertex reuse behavior of the hardware graphics pipeline implementation on current GPUs can be found in Kerbl et al. [2018]. The source code for our experiments and test applications is publicly available at https://github.com/GPUPeople/vertex_reuse.

ACKNOWLEDGMENTS

This research was supported by the Max Planck Center for Visual Computing and Communication, by the German Research Foundation (DFG) grant STE 2565/1-1, and the Austrian Science Fund (FWF) grant I 3007. The Witcher game © CD PROJEKT S.A.

REFERENCES

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

Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. 1996. Simplification Envelopes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. ACM, New York, NY, USA, 119–128. https://doi.org/10.1145/237170.237220

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

Matthew Eldridge, Homan Igehy, and Pat Hanrahan. 2000. Pomegranate: A Fully Scalable Graphics Architecture. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 443–454. https://doi.org/10.1145/344779.344981

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.

Ulrich Haar and Sebastian Aaltonen. 2015. GPU-Driven Rendering Pipelines. SIGGRAPH 2015: Advances in Real-Time Rendering in Games Talk.

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

Michael Kenzel, Bernhard Kerbl, Dieter Schmalstieg, and Markus Steinberger. 2018. A High-Performance Software Graphics Pipeline Architecture for the GPU. *ACM Trans. Graph.* 37, 4, Article 140 (Nov. 2018), 15 pages. https://doi.org/10.1145/3197517.3201374

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 (Aug. 2018), 16 pages. https://doi.org/10.1145/3233302

Jon M Kleinberg. 2000. Navigation in a small world. *Nature* 406, 6798 (2000), 845.

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

Christoph Kubisch and Pierre Boudier. 2016. GPU-Driven Rendering. GTC Talk.

Samuli Laine and Tero Karras. 2011. High-performance Software Rasterization on GPUs. In *Proc. High Performance Graphics (HPG '11)*. 79–88.

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

Fang Liu, Meng-Cheng Huang, Xue-Hui Liu, and En-Hua Wu. 2010. FreePipe: A Programmable Parallel Rendering Architecture for Efficient Multi-fragment Effects. In *Proc. I3D (I3D '10)*. 75–82.

Charles Loop. 1987. *Smooth Subdivision Surfaces Based on Triangles.* Ph.D. Dissertation.

Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. 1994. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.* 14, 4 (July 1994), 23–32. https://doi.org/10.1109/38.291528

NVIDIA. 2016. *CUDA C Programming Guide.* NVIDIA Corporation.

Anjul Patney, Stanley Tzeng, Kerry A. Seitz, Jr., and John D. Owens. 2015. Piko: A Framework for Authoring Programmable Graphics Pipelines. *ACM Trans. Graph.* 34, 4, Article 147 (July 2015), 13 pages. https://doi.org/10.1145/2766973

Karl Pearson. 1905. The problem of the random walk. *Nature* 72, 1867 (1905), 342.

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

Martin Sattlecker and Markus Steinberger. 2015. Reyes Rendering on the GPU. In *Proceedings of the 31st Spring Conference on Computer Graphics (SCCG '15)*. ACM, New York, NY, USA, 31–38. https://doi.org/10.1145/2788539.2788543

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

Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. 2012. Softshell: Dynamic Scheduling on GPUs. *ACM Trans. Graph.* 31, 6, Article 161 (Nov. 2012), 11 pages. https://doi.org/10.1145/2366145.2366180

Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU. *ACM Trans. Graph.* 33, 6, Article 228 (Nov. 2014), 11 pages. https://doi.org/10.1145/2661229.2661250

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

Graham Wihlidal. 2016. Optimizing the Graphics Pipeline with Compute. GDC Talk.

Kun Zhou, Xin Huang, Weiwei Xu, Baining Guo, and Heung-Yeung Shum. 2007. Direct Manipulation of Subdivision Surfaces on GPUs. *ACM Trans. Graph.* 26, 3, Article 91 (July 2007). https://doi.org/10.1145/1276377.1276491