High-Performance Software Rasterization on GPUs

Samuli Laine Tero Karras

NVIDIA Research*

Abstract

In this paper, we implement an efficient, completely software-based graphics pipeline on a GPU. Unlike previous approaches, we obey ordering constraints imposed by current graphics APIs, guarantee hole-free rasterization, and support multisample antialiasing. Our goal is to examine the performance implications of not exploiting the fixed-function graphics pipeline, and to discern which additional hardware support would benefit software-based graphics the most.

We present significant improvements over previous work in terms of scalability, performance, and capabilities. Our pipeline is malleable and easy to extend, and we demonstrate that in a wide variety of test cases its performance is within a factor of 2–8x compared to the hardware graphics pipeline on a top of the line GPU.

Our implementation is open sourced and available at *http://code.google.com/p/cudaraster/*

1 Introduction

Today, software rasterization on a CPU is mostly a thing of the past because ubiquitous dedicated graphics hardware offers significantly better performance. In the early times, the look of GPU-based graphics was somewhat dull and standardized because of the lack of programmability, but currently most of the stages of the graphics pipeline are programmable. Unfortunately, it looks like the increase in programmability has hit a limit, and further freedom would require the capability of changing the structure of the pipeline itself. This, in turn, is firmly rooted in the graphics-specific hardware of the GPU, and not easily changed.

The hardware graphics pipeline is a wonderfully complicated piece of design and engineering, guaranteeing that all kinds of inputs are processed efficiently in a hardware-friendy fashion. On the flip side, the complexity means that breaking the meticulously crafted structure almost certainly results in a disaster, making an incremental change towards a more flexible pipeline difficult. Fortunately, a modern GPU does not consist solely of graphics-specific units. A large portion of the work in the graphics pipeline—most notably the execution of vertex and fragment shaders but also the other programmable stages—is performed by the programmable shader cores. Most of the responsibilities of the remaining hardware revolve around data marshaling and scheduling, e.g., managing primitive FIFOs and frame buffer caches in on-chip memory, fetching shader input data into local memory prior to shader execution, triggering shader executions, running ROP, and so on. The graphics-specific units also perform a number of mostly trivial calculations such as edge and plane equation construction and rasterization, and some non-trivial but not as performance-critical ones such as clipping.

*e-mail: {slaine,tkarras}@nvidia.com

Since NVIDIA introduced CUDA [2007], the programmable shader cores have also been exposed for general-purpose programs. The overall architecture of a GPU is still chiefly optimized for running the graphics pipeline, which has implications to the kind of programs that can be run efficiently. As such, one could expect to obtain a decent performance from a purely software-based GPU graphics pipeline. The main question is how expensive it becomes to handle the duties of the graphics-specific hardware units in software. If we only consider raw FLOPs the situation does not look too grim, because with today's complex shaders, the calculations performed by the hardware graphics pipeline constitute only a small part of the overall workload. Data marshaling and scheduling is a more likely source of inefficiency, but we can expect the high memory bandwidth and exceptionally good latency-hiding capabilities of GPUs to offer some help.

In this paper, our mission is to construct a complete pixel pipeline, starting from triangle setup and ending at ROP, using only the programmable parts of the GPU. We employ CUDA for this task because it offers the lowest-level public interface to the hardware. In order to capture the essential challenges in the current graphics pipeline and to avoid oversimplifying the task, we obey several constraints imposed by current graphics APIs. Unlike previous approaches, we strictly enforce the rendering order, therefore enabling order-dependent operations such as alpha blending, and producing deterministic output which is important for verification purposes. Furthermore, we guarantee hole-free rasterization by employing correct rasterization rules.

Goals Our endeavor has multiple goals. First, we want to establish a firm data point of the performance of a state-of-the-art GPU software rasterizer compared to the hardware pipeline. We maintain that only a careful experiment will reveal the performance difference, as without an actual implementation there are too many unknown costs. Second, constructing a purely software-based graphics pipeline opens the opportunity to augment it with various extensions that are impossible or infeasible to fit in the hardware pipeline (without hardware modifications, that is). For example, programmable ROP calculations, trivial non-linear rasterization (e.g., [Gascuel et al. 2008]), fragment merging [Fatahalian et al. 2010], stochastic rasterization [Akenine-Möller et al. 2007] with decoupled sampling [Ragan-Kelley et al. 2011], etc., could be implemented as part of the programmable pipeline.

Thirdly, by identifying the hot spots in our software pipeline, we hope to illuminate future hardware that would be better suited for fully programmable graphics. The complexity and versatility of the hardware graphics pipeline does not come without costs in design and testing. In an ideal situation, just a few hardware features targeted at accelerating software-based graphics would be enough to obtain decent performance, and the remaining gap would be closed by faster time-to-market and reduced design costs.

2 Previous Work

The main use of GPUs is doing rasterization using the hardware graphics pipeline. The steady increase in GPU programmability

has sparked research of more exciting rendering paradigms such as ray tracing, first by painstakingly crafting the algorithms to fit the hardware graphics pipeline (e.g., [Purcell et al. 2002]), and later in less contrived ways through more direct programming interfaces (e.g., [Aila and Laine 2009]). There are long-standing limitations in the hardware rasterization pipeline, e.g., non-programmable blending, that restrict the set of algorithms that can benefit from hardware acceleration. Despite this, there have been few attempts to perform the entire rasterization process using a software graphics pipeline, which would allow complete freedom in this sense.

FreePipe [Liu et al. 2010] is a software rasterization pipeline that focuses on multi-fragment effects. Scheduling is very simple: each thread processes one input triangle, determines its pixel coverage and performs shading and blending sequentially for each pixel. There are numerous limitations in this approach that make it unsuitable for our purposes. Obviously, ordering cannot be retained unless only one triangle is processed at a time, but this would waste most of GPUs resources. Even if ordering constraints are dropped, highly variable number of pixels in each triangle leads to poor thread utilization. Finally, if the input consists of a handful of large triangles, e.g., in a post-processing pass where the entire screen is covered by two triangles, it is not possible to employ a large number of threads.

Because of the way the frame buffer operations are implemented using global memory atomics, FreePipe can support only 32-bit wide data on current GPUs. This means that all per-pixel data, i.e., both color and depth, has to fit in the 32-bit value. Conflicting cases where depth and color of two fragments are equal may be missed due to a race condition.

The performance of FreePipe is excellent when there are many small, fairly homogeneously sized triangles, and can even exceed the performance of the hardware graphics pipeline in certain cases. Despite this, our software pipeline is more capable than FreePipe, and in most cases more efficient as well, as demonstrated in Section 6.

Loop and Eisenacher [2009] describe a GPU software renderer for parametric patches. In their system, patches are subdivided hierarchically until a size threshold is reached, after which they are binned into screen tiles using global memory atomics. After binning, patches are further subdivided into a grid of 4×4 samples, and the resulting quads are rasterized in a pixel-parallel fashion so that each thread processes one pixel of the tile.

Larrabee [Seiler et al. 2008] is a hardware architecture that targets efficient software rasterization. Its sort-middle rasterization pipeline is similar to ours, but the only performance results are from synthetic simulations due to lack of physical hardware. The paper explicitly mentions that the simulations measure computational speed, unrestricted by memory bandwidth. This may cause inaccuracies in the results, as in a real-world situation the DRAM latency and bandwidth are rarely negligible.

3 Design Considerations

Graphics workloads are non-trivial in many ways. Each incoming triangle may produce a variable number of fragments, the exact number of which is unknown before the rasterization is complete. The number of fragments can vary wildly between different workloads, and also within a single batch of triangles. Approximately half of incoming triangles are usually culled, producing no fragments at all. Occasionally, a visible triangle may cross the near clip plane or the guardband-extended side clip planes, necessitating clipping that may produce between zero and seven sub-triangles.

Furthermore, the triangles need to be rasterized in the order they arrive, as mandated by all current graphics APIs. The ordering restriction guarantees deterministic output in case of equal depth, and enables algorithms such as alpha blending. It is possible to lift the ordering constraints in certain specific situations, for example when rendering a shadow map, but in most rendering modes the orderning has to be preserved.

3.1 Target Platform

In this paper, we target NVIDIA Fermi architecture, and more specifically the GTX 480 model that offers the highest computational power and memory bandwidth in the GeForce 400 series. The GF100 Fermi GPU in GTX 480 has 15 SMs (streaming multiprocessors) that can each hold at most 48 warps, i.e., groups of 32 threads that are always mutually synchronized. The warps are logically grouped into CTAs (cooperative thread arrays), i.e., thread blocks. Each CTA can synchronize its warps efficiently, and all its threads have access to a common shared memory storage, allowing fast communication. For a more comprehensive description of the execution model, we refer the reader to CUDA documentation [NVIDIA 2007].

In GF100, each SM has a local 64 KB SRAM bank that is used for L1 cache and shared memory. Shared memory size can be configured to either 16 KB or 48 KB, and the L1 cache occupies the rest of the space. We use the 48 KB shared memory option, because this maximizes the amount of fast memory available to the threads. The L2 cache is 768 KB in size, and it is shared among all SMs. The hardware graphics pipeline is able to pin portions of L2 memory for use as on-chip queues between pipeline stages, but in compute mode this is not possible.

Texture units are accessible in compute-mode programs, and they can be used for performing generic memory fetch operations. The texture cache is separate from L1 and L2 caches, and therefore performing a portion of memory reads as texture fetches maximizes the exploitable cache space. A texture fetch has longer latency than a global memory read, but the texture unit is able to buffer more requests than the rest of the memory hierarchy.

Atomic operations can be performed both in global memory and in shared memory. As can be expected, global memory atomics have significantly higher latency than shared memory atomics. In general, shared memory accesses are more efficient than global memory accesses, which favors algorithms that can utilize a small, fast local storage space. The graphics pipeline utilizes dedicated ROP (raster operation) units that perform blend operations and frame buffer caching, allowing SM to perform frame buffer writes in a fire-and-forget fashion. In compute mode the ROP units are not accessible, which forces us to carry out these operations in SM.

As in any massively parallel system, the best performance is obtained by minimizing the amount of global memory traffic, by minimizing the amount of expensive synchronization operations, and by ensuring that as many threads as possible are executable at any given time, i.e., not disabled or pending synchronization or memory operation. In our target platform this translates to using shared memory instead of global memory where possible, avoiding synchronization across CTAs, and keeping as many threads active as possible by avoiding execution divergence.

3.2 Buffering and Memory

The hardware graphics pipeline buffers as little data as possible and keeps it in on-chip memories. This behavior cannot be replicated in software as-is, because the graphics-specific buffer and queue

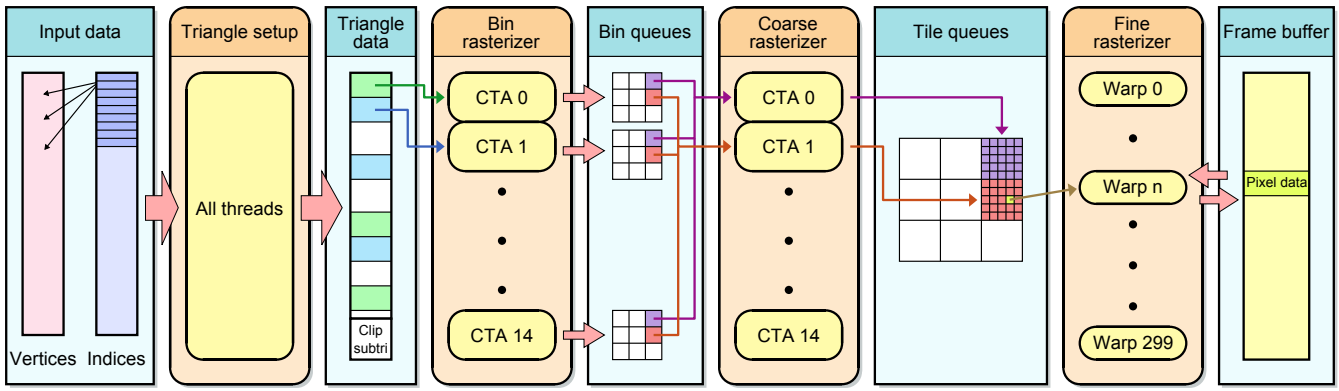


Figure 1: *A high-level diagam of our software rasterization pipeline. The structure is discussed in detail in Sections 4 and 5. Triangle setup stage processes the input triangles and produces one triangle data entry for each input triangle. Bin rasterizer CTAs read the entries in large chunks and each CTA produces a queue of triangles for each bin to avoid synchronization when writing. These queues are merged in coarse rasterizer, where each CTA processes one bin at a time and produces per-tile triangle queues. The per-tile queues are processed by the fine rasterizer, where each warp processes one tile at a time.*

management hardware is inaccessible, and emulating it in software would be costly. In addition, launching different types of shaders on-demand is not possible. Fortunately, the GPU memory architecture is very efficient and offers a lot of bandwidth even to off-chip DRAM. Therefore, as long as the amount of data being transferred is not excessive, we can simply stream the inputs and outputs of each pipeline stage through DRAM without devastating overhead. This has the advantage of enabling a chunking, or sort-middle [Molnar et al. 1994], architecture, where data locality is captured early in the pipeline and exploited in later stages.

In a sort-middle architecture, the amount of frame buffer traffic is minimal. After enough, optimally all, of the primitives touching a screen-space tile are buffered beforehand, we can transfer the frame buffer tile to on-chip memory once, perform the per-pixel operations, and submit the tile back into DRAM. In our implementation, non-MSAA modes function like this, but with MSAA the amount of data per tile is too large to be kept in on-chip memory and there is therefore significantly more frame buffer traffic. Inaccessible to us, the hardware ROP employs frame buffer compression to reduce the amount of off-chip memory traffic. In cases where we transfer the tile only once, we expect the benefits of compression to be negligible.

To maximize the benefits of a sort-middle architecture, we execute the entire pipeline from start to finish for as large portion of input as possible, and store the entire intermediate buffers in DRAM. The batch sizes can be maximized by grouping together draw calls for the same render target, and choosing the shader in the fine rasterizer stage based on a per-triangle shader ID. This avoids the need to flush the pipeline except when absolutely necessary, for example when binding a render target to a texture.

The worst-case output of a pipeline stage is usually so large compared to average output that it does not make sense to allocate buffer space for the worst-case situation. Instead, we detect when a buffer runs out of space, so that the batch can be aborted and the data can be submitted again in smaller batches. This is not a particularly elegant solution, but works very well in practice as the workload in typical content rarely varies much from frame to frame. In addition, by monitoring the data sizes, the overruns can usually be prevented in advance.

3.3 Queues and Synchronization

To minimize the risk of execution stalls, we have decided to avoid any inter-CTA synchronization when reading from and writing to queues. This design choice has a large impact on the implementation of the pipeline. First of all, writing to any queue must be performed by a single CTA that can synchronize efficiently internally. Furthermore, one CTA can only write to a limited number of queues efficiently. To perform efficient parallel queue writes, we need to use shared memory for collecting the elements to be written to each queue and calculating an output offset for each element. The amount of shared memory therefore limits the maximal data expansion factor of a pipeline stage.

To utilize the available memory efficiently, we allocate memory dynamically to queues as they are written to, and each allocation unavoidably requires one globally atomic operation. To minimize the number of allocations, our queues consist of segments which are equally-sized, contiguous memory ranges, and a queue is a linked list of references to these segments. Allocating memory is necessary only when the last segment of a queue becomes full.

3.4 Rasterization

Fine rasterizer, i.e., the unit that determines which samples of a small pixel stamp are covered by the triangle, is traditionally held as one of the crown jewels of the hardware graphics pipeline. The parallel evaluation of the edge functions at sampling points for multiple samples using custom ALUs is extremely area- and power-efficient. However, if we consider all of the calculations that are performed per-fragment, we can legitimately suspect that this is not a major portion of the overall workload. Therefore, we shall not let the lack of access to the hardware rasterizer dispirit us, and expect that a properly optimized software rasterizer will provide sufficient performance.

4 Pipeline Structure

In this section, we describe the high-level design of our pipeline. We start with an overview, and continue by discussing in detail how data is passed between stages and how each stage is parallelized. The structure of the pipeline is illustrated in Figure 1.

Our pipeline consists of four stages: triangle setup, bin rasterizer, coarse rasterizer, and fine rasterizer. Each stage is executed as a separate CUDA kernel launch. We exclude the vertex shader from consideration, as it can be trivially executed as a one-to-one mapped CUDA kernel without any ordering constraints. Triangle setup performs culling, clipping, snapping, and calculation of plane equations. Bin rasterizer and coarse rasterizer generate, for each screen-space tile, a queue that stores the triangle IDs that overlap the tile. The reasons for splitting this operation in two stages are discussed below. The last stage, fine rasterizer, processes each frame buffer tile, computing exact coverage for the overlapping triangles, executing the shader, and performing ROP.

4.1 Dataflow

The input of triangle setup is a compact array of triangles. Likewise, the output is an array of triangles, but there is no direct one-to-one mapping between input and output. In the majority of cases, each input triangle generates either zero or one output triangles due to culling, and in rare cases the clipper may produce many output triangles. We could either produce a continuous, compacted triangle array as output, or artificially keep one-to-one relationship between input and output, and compact/expand the output array of set-up triangles as it is read into the next stage. Because at this point we do not have any input parallelism, i.e., there is only one input array, we have chosen to do the latter. This relieves us from ordering concerns and allows us to trivially employ the entire GPU.

After triangle setup, culling, and clipping, each triangle may generate a variable amount of work. The trivial solution is to expand each triangle directly to pixels and shade them immediately, which is what FreePipe [Liu et al. 2010] does. Because of the numerous problems related to this approach, as discussed in Section 2, we instead turn to the standard sort-middle solution, which is to divide the screen into tiles, and for each tile construct a queue of triangles that overlap it. If we wish to keep the frame buffer content of a tile in shared memory during shading, the tiles need to be fairly small. In practice, this limits us to 8×8 pixel tiles with 32-bit depth and 32-bit color in non-antialiased mode. Unfortunately, with this small tiles, even a modestly sized viewport will contain many more tiles than can be efficiently written to from a single CTA, for reasons that were outlined in Section 3.3.

Our solution is to split this part of the pipeline into two stages, bin rasterizer and coarse rasterizer. We first rasterize the triangles into bins that contain 16×16 tiles, i.e., 128×128 pixels, and after that process each bin to produce per-tile queues. By restricting the viewport size to 2048×2048 pixels, i.e., 16×16 bins, the expansion factor is limited to 256 in both stages. This is small enough to allow efficient queue writes, as detailed in Section 5.2.

4.2 Parallelization

Let us now consider how to properly employ the entire GPU in each stage. Triangle setup is trivial, because we enforce one-to-one mapping and can therefore process the input without worrying about ordering or compacting the output. The next stage, bin rasterizer, is the most complicated to parallelize. Given that there is only one input queue from the triangle setup, and the triangles in each per-bin queue need to be in input order, the obvious choices are either utilizing only one CTA or performing expensive inter-CTA synchronization before every queue write to ensure ordering.

Neither of these options is attractive. Instead, our solution is to produce as many per-bin queues as we launch bin rasterizer CTAs, so that every CTA writes to its own set of per-bin output queues. This is similar to the approach taken by Seiler et al. [2008]. This removes the need to synchronize between CTAs when writing, and each individual output queue is still internally in order. When reading the per-bin queues, the coarse rasterizer has to merge from multiple queues. The cost of merging is decreased by having the bin rasterizer process the input in large, continuous chunks. This way, the merging can be done on a per-segment basis instead of per triangle.

The coarse rasterizer can be easily parallelized by processing each bin in a separate CTA with as many threads as possible. This avoids conflicts among inputs and outputs between individual CTAs. The number of nonempty bins is typically higher than the number of concurrently executing CTAs, yielding fairly good utilization. The fine rasterizer is also trivial to parallelize by processing each tile in a single warp. As long as there are enough nonempty tiles in a frame, the entire GPU is properly utilized and each warp has an exclusive access to its inputs and outputs.

5 Pipeline Stages

We shall now examine each of the four pipeline stages in detail. For the sake of clarity, queue memory management and overrun detection are left out of the description, as well as several low-level optimizations.

5.1 Triangle Setup

The triangle setup is executed using a standard CUDA launch, and each thread is given the task of processing one triangle. Ordering is implicitly preserved, because each set-up triangle is written to the output array in the index corresponding to the input triangle.

Each input triangle is an index triplet that refers to vertex positions that are stored in a separate array created by a previously executed vertex shader. After reading the vertex positions, view frustum culling is performed, after which the vertex positions are projected into viewport and snapped to fixed-point coordinates. If any of the vertices is outside the near or far clip plane, or the AABB (axis-aligned bounding box) of the projected triangle is too large, the triangle is processed by the clipper. In this case, each of the resulting triangles is snapped and processed sequentially by the same thread, and appended into a separate subtriangle array.

Multiple culling tests are performed for each triangle. If the triangle is degenerate, i.e., has zero area, it is culled, as well as if the area is negative and backface culling is enabled. If the AABB of the triangle falls between the sample positions, we also cull the triangle. This test is very effective in culling thin horizontal and vertical triangles that often result when viewing distant axis-aligned geometry in perspective. Finally, if the AABB is small enough to contain only one or two samples, we calculate their coverage, and if no samples are covered, we cull the triangle. This ensures that for densely tessellated surfaces, we output at most one triangle per sample, which can be much fewer than the number of input triangles.

If the triangle survives the culling tests, we compute screen-space plane equations for (z/w), (u/w), (v/w), and $(1/w)$, which are all linear. We also separately store minimum (z/w) over all vertices to enable hierarchical depth culling in the fine rasterizer stage.

Our implementation is optimized for the common case where the triangle produces zero or one outputs due to culling. In these cases, the output record is self-contained, and no dynamic memory allocation is needed. If the triangle needs to be clipped, we insert the resulting subtriangles in a separate array, and the output record contains references to these. Each clipped triangle therefore requires that we reserve space from the end of the subtriangle array by a global memory atomic. However, this happens so infrequently that the cost is negligible.

5.2 Bin Rasterizer

We execute one bin rasterizer CTA per SM, each containing 16 warps and therefore 512 threads, and keep these CTAs running until all triangles are processed. We use persistent threads in a similar fashion as Aila and Laine [2009] did in context of GPU ray tracing, but for different reasons; our primary goal is to minimize the total number of per-CTA queues. Each of the 15 CTAs works independently, except when picking a batch of triangles to be processed, which is performed using a global memory atomic. To reduce the number of atomic operations, the input is consumed in large batches. The batch size is calculated based on the input size.

After acquiring an input batch, we enter the input phase where setup triangles are read from the batch. Each of the 512 threads reads one triangle setup output record, which may correspond to 0–7 triangles. A cumulative sum of triangles is calculated over all threads to determine storage position for each subtriangle in a triangle ring buffer that is stored in shared memory. This compacts away culled triangles and spreads out the subtriangles produced by the clipper. As long as the input batch is not exhausted and there are fewer than 512 triangles in the ring buffer, the input phase is repeated. Note that the triangles are stored in the ring buffer in the same order they are read from the input.

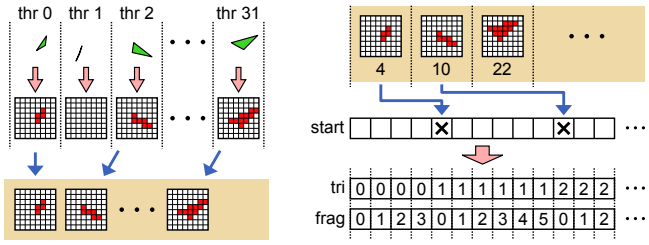
When we have collected 512 triangles, or the input batch is exhausted, we switch to rasterization phase where each thread processes one triangle. First, we determine which bins the triangle covers. Edge functions and an AABB are calculated from the snapped fixed-point vertex coordinates, and each bin covered by the AABB is checked for overlap by the triangle. If the AABB covers at most 2×2 bins, which is the most common case, we skip the overlap tests and edge function setup, and simply assume that the triangle overlaps each bin. When the AABB coverage is 1×1, 1×2, or 2×1 bins, this is in fact the correct solution as the triangle must overlap each of those bins, and in 2×2 case at least three bins are guaranteed to be covered.

We tag the overlapped bins in a bit matrix in shared memory that holds one bit per triangle per bin. With 512 triangles and a maximum of 16×16 bins, this amounts to 16 KB, consuming one third of the 48 KB available in shared memory. After the coverage of all triangles has been resolved, we synchronize the CTA and calculate the output indices for the triangles in each per-bin output queue. Each thread is responsible for writing its own triangle, so it needs to know the proper output index for each of the covered bins. We calculate this by first tallying per-bin, per-warp write counts, which are then cumulatively summed over to determine start index for each warp for each bin. When writing the triangles, the threads in each warp calculate their own output indices within the warp using intra-warp one-bit cumulative sums that are efficiently supported by the hardware.

The calculation of output indices is performed in such an order that the triangle IDs are stored in the output queues in the same order they were read in. When the input batch is finished, we flush the output queues by marking the last segments as full before proceeding to grab the next input batch. This ensures that merging the per-CTA queues on a per-segment basis, instead of per-triangle, is sufficient in the coarse rasterizer stage, because each segment corresponds to a single, continuous part of the input.

5.3 Coarse Rasterizer

Similarly to bin rasterizer, we execute one coarse rasterizer CTA per SM and keep them running until all input is processed. Each CTA has 16 warps, again amounting to 512 threads per CTA. We sort the bins from largest to smallest by each CTA, and process them in this order to minimize end-of-launch underutilization.



(a) Input phase (b) Shading phase

Figure 2: *Coverage calculation and fragment distribution in fine rasterizer. (a) In the input phase, all threads in a warp calculate coverage for one triangle, and the coverage masks are stored in a ring buffer. Empty coverage masks are compacted away. To keep track of the fragment count, a running total of fragments is also stored for each triangle. (b) In the shading phase, each thread marks the start of one triangle in a bitmask, based on fragment counts. Because empty coverage masks have been culled, no conflicts can occur. Then, based on the bitmask, each thread can calculate the indices of the triangle and the fragment it should shade.*

At any given time, each CTA works on exactly one bin, and no two CTAs can be processing the same bin. We therefore start by picking a bin to process using a global memory atomic, and similarly to bin rasterizer, enter the input phase. We read an entire segment at a time, and each time need to determine which of the 15 input queues from the bin rasterizer contains the next segment in input order. This is easily done by looking at the triangle index of the next entry of each queue, and choosing the smallest one.

When at least 512 triangles have been read, we enter the rasterization phase where each thread processes one triangle. Similarly to bin rasterizer, we determine which tiles of the bin each triangle covers, and tag these into a bit matrix in shared memory. The writing of triangles into per-tile queues is performed differently, because in coarse rasterizer there is much more variance in the number of covered tiles between triangles.

Instead of writing one triangle from each thread, we calculate the total number of writes the CTA has to perform, and distribute these write tasks evenly to all 512 threads. To perform a write, the thread has to find, based on the task index, which tile the write is targeted for, which warp is responsible for it, and finally, which triangle is in question. Each of these is implemented as a binary search over a small domain. Even though each individual write is fairly complicated, this balancing provides speedup over the simpler thread-per-triangle approach used in bin rasterizer.

5.4 Fine Rasterizer

The work in fine rasterizer is divided on a per-warp basis, and there is no communication between warps. We launch 20 warps per SM and keep them running until the frame is finished. Each warp processes its own tile, which is selected using a global memory atomic, and has an exclusive access to it. We shall first discuss the non-MSAA case, where we store the frame buffer tile in shared memory. If we are processing the first batch after a clear, we set up a cleared frame buffer tile in shared memory. Otherwise, we read the tile from DRAM. We can afford keeping per-pixel 32-bit RGBA color and 32-bit depth in shared memory, amounting to 10 KB with 8×8 pixel tiles and 20 warps.

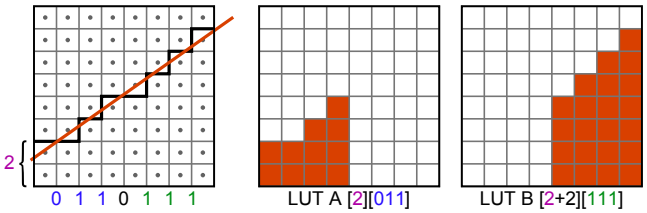


Figure 3: *Our 8×8 pixel coverage calculation is based on look-up tables. Based on relative positions of vertices, we swap/mirror each coordinate so that the slope of the edge is between 0 and 1. (a) We then determine the height of the edge at leftmost pixel column, and for each column transition we determine if the edge ascends by one pixel. This yields a string of 7 bits. (b) The coverage mask is fetched in two pieces from a look-up table. The offset for the second lookup is obtained by incrementing the first offset by the number of set bits among the first four bits. With this technique, a 8×8 pixel coverage mask can be produced in 51 assembly instructions per edge on GF100. The splitting of the look-up table is done to shrink the memory usage to 6 KB, allowing us to store the table in fast shared memory.*

The fine rasterizer is divided into two phases, first of which is the input phase (Figure 2a). We read 32 triangles in parallel from the per-tile input queue, and calculate a 64-bit pixel coverage mask for each triangle using a LUT-based approach (Figure 3). The triangle index and coverage mask are stored in a triangle ring buffer in shared memory. Triangles that cover no samples are compacted away; this can happen when a triangle falls between samples or merely grazes the tile.

We keep count of fragments, i.e., covered pixels, in the triangle ring buffer, and as soon as at least 32 fragments are available, we switch to the shading phase. We distribute the fragments to threads so that each thread processes one fragment. The fragments may come from different triangles, and the distribution is performed so that all fragments of a later triangle are given to threads with higher lane index than the fragments of earlier triangles, as illustrated in Figure 2b. We first calculate depth using the (z/w) plane equation, and kill the fragment if the depth test fails. The surviving threads continue to execute the shader and ROP. The processed triangles and fragments are removed from the ring buffer, and if fewer than 32 fragments are left, we enter the input phase again. When the entire input is processed, we write the frame buffer tile into DRAM.

Let us now examine the key components of the fine rasterizer in detail, and provide extensions to the basic scheme outlined here.

Shader We interpolate attributes based on barycentric coordinates calculated from screen-space (u/w), (v/w) and $(1/w)$ plane equations that are constructed in the triangle setup stage. After evaluating the barycentric coordinates for the shading point, we fetch the vertex attributes and interpolate them. This is a much more expensive process than what the hardware graphics pipeline uses; the issue is discussed further in Section 6.1.

ROP The method of updating the frame buffer is chosen based on the depth test and blend modes. If no depth test and no blending is performed, we simply have each thread write its results into the tile in shared memory. When there are shared memory write conflicts within the warp, the write from a thread on a higher lane, therefore containing a later triangle, will override a write from a thread on a lower lane, containing an earlier triangle. The CUDA programming guide explicitly leaves it undefined which thread will succeed in the write, but at least on GF100 the behavior is consistent and can be exploited. If this is changed in future hardware, we can fall back to a slightly less efficient scheme based on shared memory atomics.

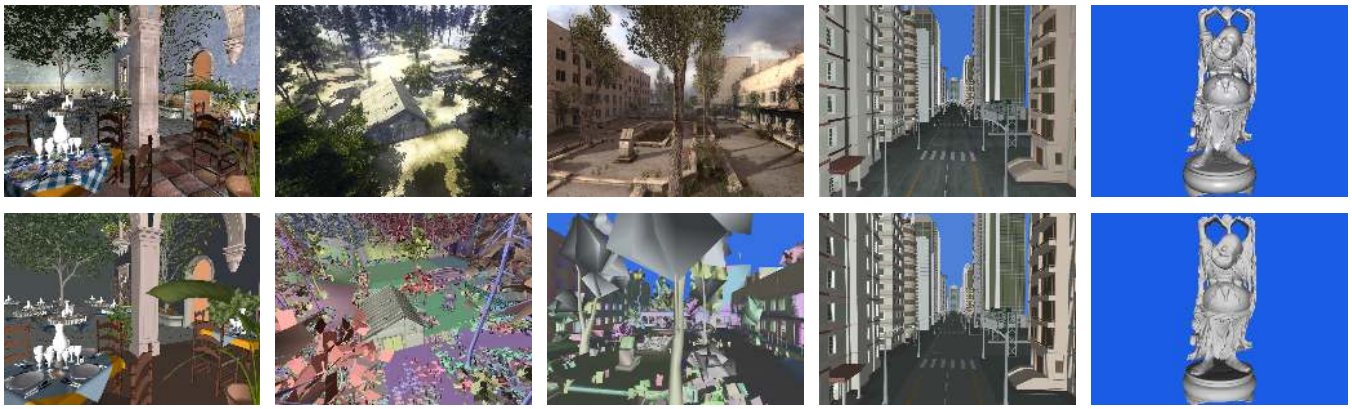
When depth test or blending is enabled, the lane ordering is reversed to make the writes of an earlier triangle prevail over a later one in conflicts, and each thread loops until its write succeeds. When depth test is enabled, this is detected by reading back the depth that was written into shared memory, and repeating as long as the depth test succeeds. Because simultaneous writes from later fragments override earlier ones in both color and depth, this somewhat surprisingly yields correct results. When depth test is disabled but blending is enabled, we use the index of the writing thread instead of depth to detect when the write is successful.

Hierarchical Z A simple way to improve performance is to perform hierarchical depth kills [Greene et al. 1993] on a per-triangle level. With the typical depth ordering, this is achieved by maintaining z_{max}, the farthest depth value found in the current frame buffer tile. By comparing this against the minimum triangle depth, computed in the triangle setup stage, we can discard triangles that are entirely behind the surfaces already drawn in the tile. We calculate z_{max} in the fine rasterizer using warp-wide reduction whenever we fetch a tile from DRAM or overwrite a depth value that equals the current z_{max}. This avoids having to store z_{max} in off-chip memory while minimizing the number of unnecessary updates.

Quad-pixel shading Performing a dependent mipmapped or anisotropic texture fetch requires knowledge of the derivatives of the texture coordinates. In hardware graphics pipeline, these are calculated by grouping all shaded fragments into *quads*, i.e., aligned groups of 2×2 pixels, and the texture unit automatically estimates the derivatives by subtracting the texture coordinates of adjacent pixels in such group. Unfortunately, this functionality is not available in compute mode, and we therefore need to do this programmatically. If the fragment shader requires derivatives, we expand the pixel coverage mask to include all pixels in even partially covered 2×2 pixel quads, and therefore have 8 such quads being shaded in a warp. Taking the derivatives is performed by subtracting texture coordinates through shared memory. Note that non-dependent texture fetches do not usually require quad-pixel shading, as analytic derivatives of the texture coordinate attributes can be evaluated directly based on the barycentric coordinate plane equations.

MSAA In the hardware graphics pipeline, multisample anti-aliasing [NVIDIA 2001] comes almost without extra cost. The major burden is on rasterizer, which has to calculate coverage for multiple samples per pixel, and on ROP that has to perform blending for multiple samples in case it is enabled. Both of these units are implemented using dedicated hardware, and can therefore be well optimized for these tasks.

In our software pipeline, we defer the per-sample coverage calculation as far in the pipeline as possible. We replace the evaluation of coverage at the center of the pixel by a conservative triangle-vs-pixel test. This allows us to avoid the processing of a pixel in case it is not overlapped by the triangle at all. The early per-pixel depth test is similarly replaced by a conservative depth test against per-pixel z_{max} value that we keep in shared memory. For the surviving pixels, we determine the coverage of each sample and execute shader if any of the samples are covered. Each thread then executes the ROP for each sample sequentially. Having each thread process the same sample ID at the same time makes it easy to detect conflicting writes into the same sample.



SAN MIGUEL, 189MB
5.44M tris, 25% visible
2.4 pixels / triangle

JUAREZ, 24MB
546K tris, 37% visible
14.6 pixels / triangle

STALKER, 11MB
349K tris, 41% visible
14.1 pixels / triangle

CITY, 51MB
879K tris, 21% visible
16.3 pixels / triangle

BUDDHA, 29MB
1.09M tris, 32% visible
1.4 pixels / triangle

Figure 4: *Top row: original test scenes with proper shading and textures. Bottom row: Gouraud-shaded versions used in the measurements. The memory footprint numbers represent the size of the raw geometry data (32 bytes per vertex and 12 bytes per triangle). The percentage of visible triangles accounts for the effects of backface culling and view frustum culling, and was computed as an average over 5 camera positions. Screen-space triangle area is an average over the visible triangles in 1024×768 resolution.*

Unfortunately, we cannot any more store the entire tile in shared memory, but are forced to execute the ROP directly on global memory. We still use shared memory for storing the per-pixel z_{max} that is used for culling entire pixels. Also, we perform the serialization of the writes in correct order by storing and reading back the writing thread index in a per-pixel shared memory location.

6 Results and Discussion

We evaluated the performance of our software pipeline on GeForce GTX 480 with 1.5 GB of RAM, installed in a PC with 2.80 GHz Intel Core i7 CPU and 12 GB of RAM. The operating system was Windows 7, and we used the public CUDA 3.2 driver. For comparison, we ran the same test cases on the hardware pipeline using OpenGL, as well as on our implementation of FreePipe [Liu et al. 2010] optimized for the hardware used. For maximum performance in FreePipe, we maintain only one 32-bit color/depth entry per pixel.

Even though the total rendering time tends to be dominated by the cost of vertex and fragment shading in modern real-time content, we are mainly interested in raw rasterization performance for two reasons. First, the complexity of shader programs is highly application-dependent, and choosing a representative set of sufficiently complex shaders is hardly a trivial task. Second, shaders are executed by the same hardware cores in both pipelines, so their performance is essentially the same except that the hardware pipeline is able to perform rasterization in parallel with shading. Thus, we employ simple Gouraud shading, i.e., linear interpolation of vertex colors, in our benchmarks, and assume that the input geometry has already been processed by a vertex shader. The format of the input data is the same for all comparison methods, and consists of 4-component floating point positions and colors per vertex accompanied by three 32-bit vertex indices per triangle. Backface culling is enabled in all tests, and all results are averages over five different camera positions.

Test scenes used in the measurements are shown in Figure 4. The top row shows the scenes with proper shading and textures, while the bottom row shows the Gouraud-shaded versions. Two of the scenes, JUAREZ and STALKER, were chosen to represent game content. They were constructed from DirectX geometry captures from Call of Juarez (Techland) and S.T.A.L.K.E.R.: Call of Pripyat (GSC Game World). SAN MIGUEL is a test scene included in PBRT, and includes a lot of vegetation that consists of very small triangles.

Scene	Resolution	HW	Our (SW)	FreePipe (FP)	SW:HW ratio	FP:SW ratio
SAN MIGUEL	512×384	5.37	7.82	130.14	1.46	16.65
	1024×768	5.43	9.48	510.20	1.74	53.84
	2048×1536	5.86	15.44	1652.52	2.64	107.06
JUAREZ	512×384	0.59	2.71	5.34	4.56	1.97
	1024×768	0.67	3.28	18.63	4.87	5.69
	2048×1536	1.03	7.06	72.45	6.84	10.26
STALKER	512×384	0.31	1.81	23.47	5.91	12.96
	1024×768	0.39	2.31	92.73	5.96	40.14
	2048×1536	0.67	5.41	386.07	8.10	71.36
CITY	512×384	0.93	2.16	64.56	2.32	29.88
	1024×768	1.04	3.13	251.86	3.01	80.54
	2048×1536	1.42	6.79	1032.83	4.77	152.13
BUDDHA	512×384	1.06	2.09	2.14	1.98	1.02
	1024×768	1.07	2.66	3.08	2.50	1.16
	2048×1536	1.11	4.01	6.96	3.62	1.73

Table 1: *Performance comparison between the hardware pipeline, our software pipeline, and FreePipe. The values are in milliseconds and represent the total rendering time, excluding vertex shader and buffer swaps. The SW:HW ratio shows the hardware performance compared to our pipeline (higher values mean that the hardware is faster). Similarly, the FP:SW ratio shows the performance of FreePipe compared to ours (higher values mean that our pipeline is faster). All measurements were performed with depth test enabled, without MSAA or blending.*

Table 1 compares the performance of the three pipelines in the non-MSAA case, with depth test enabled and blending disabled. We see that our pipeline is generally 1.5–8x slower than the hardware, but in most cases a magnitude or two faster than FreePipe. The hardware pipeline scales well with increasing resolution, which indicates that per-triangle operations such as attribute fetch and triangle setup are relatively costly compared to rasterization and per-fragment operations. This is especially true for SAN MIGUEL which places a high burden on triangle setup. Our performance is

Render mode	MSAA mode	San Miguel			Juarez		
		HW	SW	Ratio	HW	SW	Ratio
Depth test, no blend	1	5.43	9.48	1.74	0.67	3.28	4.87
	2	5.55	15.21	2.74	0.78	5.60	7.16
	4	5.75	20.62	3.58	0.96	6.78	7.09
	8	6.28	28.24	4.50	1.37	9.38	6.84
No depth test, alpha blend	1	5.44	7.99	1.47	0.71	3.22	4.54
	2	5.57	14.68	2.63	0.85	5.89	6.91
	4	5.77	20.77	3.60	1.10	7.47	6.81
	8	6.38	29.69	4.65	1.81	10.64	5.89
Depth only, no color write	1	5.37	6.80	1.27	0.65	2.62	4.02
	2	5.46	11.49	2.11	0.75	4.46	5.93
	4	5.59	16.24	2.91	0.91	5.49	6.06
	8	5.98	23.24	3.89	1.26	7.77	6.16

Table 2: *Effect of rendering and antialiasing modes. The values are frame times in milliseconds, and the Ratio column shows the hardware performance compared to ours. All measurements were done in 1024×768 resolution.*

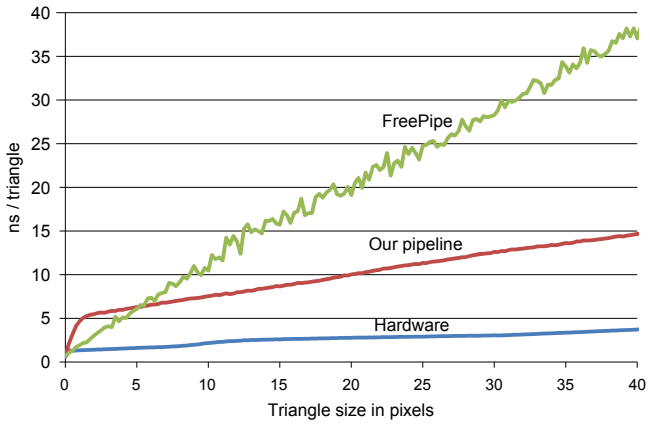


Figure 5: *Effect of triangle size on rendering time in a synthetic test case with equally sized triangles. Horizontal axis is the triangle area in pixels, and vertical axis is the average rendering time per triangle in nanoseconds. The results are for flat-shaded triangles without multisampling, depth test, or blending. Note that this is the absolute best case for FreePipe, and its performance is often reduced by variance in triangle sizes (see Table 1).*

much more sensitive to resolution, which is explained by our lower fill rate. FreePipe performs reasonably well in Buddha because all triangles are of roughly the same size. However, mixing triangles of different sizes quickly leads to serious underutilization of the GPU due to the lack of load balancing.

To gain further insight into our performance with various triangle sizes, we constructed a synthetic test case of equally-sized triangles organized into multiple screen-sized layers. Figure 5 shows a sweep of the average per-triangle cost as a function of triangle size. With triangles covering 10 pixels, the average rendering rate is 470 Mtris/s for hardware, 130 Mtris/s for our pipeline, and 100 Mtris/s for FreePipe. With very large triangles the pixel fill rates approach 12 Gpix/s, 3 Gpix/s, and 1 Gpix/s, respectively. Note that this test case is ideal for FreePipe, since all threads perform exactly the same amount of computation. For example, if the input consisted of interleaved triangles of two different sizes, the performance of FreePipe would be dictated by the larger ones.

Table 2 explores the effect of different rendering and antialiasing modes in two of the test scenes. In the hardware pipeline, increasing

Statistic	unit	San Miguel	Juarez	Stalker	City	Buddha
Tri setup	ms	4.79	0.77	0.47	0.93	1.16
Bin raster	ms	1.45	0.36	0.21	0.26	0.45
Coarse raster	ms	1.46	0.76	0.63	0.76	0.56
Fine raster	ms	1.78	1.38	1.00	1.17	0.50
Tri data	MB	420.0	42.2	26.9	67.9	84.0
Bin queues	MB	4.0	1.5	1.2	0.9	2.0
Tile queues	MB	4.4	2.9	2.2	2.2	1.5
Outside frustum	%	42.4	28.5	12.8	26.4	27.9
Backfacing	%	32.3	34.2	46.2	52.9	40.0
Between samples	%	17.7	6.7	17.3	14.8	16.2
Surviving	%	7.7	30.5	23.6	5.9	15.9
Tris / tile		71.0	42.4	25.8	24.0	60.4
Frags / tile		265.2	248.8	180.0	227.4	77.0
Hier. Z kill	%	22.5	14.9	37.1	19.4	0.0
Early Z kill	%	37.6	48.5	40.8	39.7	0.0
ROP rounds		1.19	1.21	1.10	1.07	1.12

Table 3: *Statistics from rendering the test scenes with our pipeline. See the text for details.*

the number of MSAA samples has more or less the same effect on performance as increasing the resolution. Our performance drops significantly as soon as MSAA is enabled because we can no longer cache frame buffer tiles in shared memory. The slowdown when increasing the number of samples is mainly explained by the increase in frame buffer DRAM traffic. Unlike on hardware, disabling color writes improves the performance of our pipeline by 20–30%. This is because we can completely skip the barycentric plane equation setup and attribute interpolation, both of which are relatively costly in software.

Table 3 lists a number of statistics from rendering the test scenes with our pipeline in 1024×768 resolution with depth test enabled and blending disabled, without MSAA. The first group of statistics shows a breakdown of the total rendering time into individual stages. As expected, San Miguel and Buddha are dominated by triangle setup, which corresponds to 51% and 43% of the total rendering time, respectively. In scenes with larger triangles the rendering time is dominated by the fine rasterizer.

The second group lists the total memory footprint of the intermediate buffers. Tri data is the size of the data array produced by the triangle setup stage, which is approximately 1–3 times larger than the raw scene data in our test cases. With higher number of vertex attributes, this ratio would decrease because the output of the triangle setup stage does not depend on the number of attributes. Bin queues and Tile queues are the sizes of the triangle ID queues produced by bin and coarse rasterizer stages, respectively. Contrary to what one might expect, these are almost negligible compared to the scene data. This is because the majority of triangles tend to intersect just one bin and only a handful of tiles, and we need to store only a single 32-bit triangle index per entry.

The third group shows a breakdown of triangles culled by triangle setup, and the fourth group lists a few interesting statistics for the fine rasterizer stage. The culling of small triangles that fall between sample positions is surprisingly effective in all of the test scenes. In San Miguel, for example, the number of triangles passed down the pipeline decreases by a factor of 3.3. The average number of triangles per tile dictates the efficiency of the input phase of the fine rasterizer. The triangles are consumed in batches of 32, and some of the threads remain idle for batches that contain fewer triangles. There is a similar relation between the number of fragments per tile and the efficiency of the shading phase.

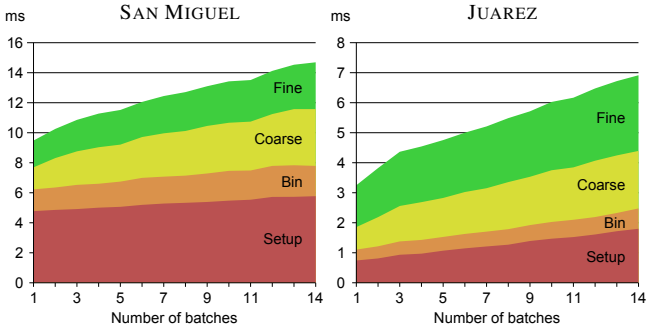


Figure 6: *The effect on execution time when input is split into multiple equally-sized batches. The stacked area graphs show the execution times of the four pipeline stages in* SAN MIGUEL *and* JUAREZ *rendered with depth testing, no blending, and no MSAA in 1024×768 resolution.*

Hier. Z kill indicates the percentage of triangles culled by the hierarchical Z test in the input phase, and Early Z kill indicates fragments culled at the beginning of the shading phase. As expected, both are relatively high in all test scenes except BUDDHA, which lacks depth kills because the triangles happen to be drawn in back-to-front order. Finally, ROP rounds indicates the average number of times the ROP loop is executed for a single batch of 32 fragments. Due to conflicting writes to the same pixel, the loop may get iterated up to 32 times. However, conflicts are very rare in practice.

We can see that the number of triangles intersected by each tile is relatively low even though we render the entire scene in a single batch. It is therefore interesting to know how much our performance depends on the batch size. Figure 6 shows a sweep of the total rendering time in two test scenes with input split into various numbers of equally-sized batches. As expected, processing the input in small batches decreases performance because small batches do not utilize the GPU as efficiently as large ones, and constant per-batch costs are thus comparatively higher.

6.1 Scalability and Ideas for Hardware Extensions

Let us briefly consider how our implementation scales as various parameters are modified. The viewport size is limited to 2048×2048 pixels in our pipeline for two reasons. First, we can only write to a limited number of queues per CTA in bin and coarse rasterizers, and secondly, for higher resolutions we would need to decrease the number of subpixel bits if we wish to keep using 32-bit arithmetic. Even now, our subpixel resolution of 4 bits is lower than in the hardware graphics pipeline (8 bits). To support larger viewports without losing subpixel precision, many of the intermediate data structures, e.g., vertex positions and plane equation coefficients, would need to use higher precision, and more importantly, internal calculations in the rasterizers would need to performed using 64-bit arithmetic.

Attribute interpolation is one of our biggest weaknesses compared to the hardware graphics pipeline. We need to programmatically fetch vertex attributes and perform interpolation in the fragment shader. The hardware pipeline avoids this by calculating attribute plane equations in hardware before launching the fragment shader [Lindholm et al. 2008]. Furthermore, dedicated hardware is used for performing the interpolation arithmetic. Enabling similar performance in a software-based graphics pipeline is not trivial. Enabling access to the interpolation hardware would not be enough, as the fetching of data is also an a major source of inefficiency. To what extent this could be battled by prefetching data in software is an interesting question.

Discounting attribute interpolation, our shader performance should be on par with the hardware graphics pipeline, as mostly the same code is executed in both cases. We hypothesize that in cases where the current hardware pipeline is particularly inefficient, we could provide better performance through higher thread utilization. For example, with a discard-heavy shader, the fine rasterizer could be extended to compact the fragments after discards, and thereby obtain better thread utilization for the shading computations. Also, when the triangles are very small and quad derivatives are not required, we do not suffer from the unnecessary expansion to 2×2 pixel quads. This has been identified as a potentially major source of inefficiency [Fatahalian et al. 2010]. In terms of triangle counts our scalability is fairly good, and especially the culling of small triangles is extremely efficient compared to the hardware graphics pipeline.

There are a few hot spots in our software pipeline that would benefit from fixed-function hardware acceleration. The pixel coverage calculation currently takes approximately 160 instructions per triangle, which corresponds to roughly 40% of the hardware rasterizer throughput. If a coverage calculation unit were integrated with the SM, we could effectively nullify this cost, bringing approximately 3–7% speedup. Another time-consuming operation is finding the nth set bit in the coverage mask, which could be easily accelerated with a custom instruction. This would provide approximately 2–4% speedup with very little design effort.

Increasing the performance of shared memory atomics would help almost all of our pipeline stages, and provide perhaps 5–10% speedup. Even better, but much more involved, would be efficient hardware support for queue read and write operations, because that would eliminate most of the code in the bin and coarse rasterizers. It is however unclear what the semantics should be, and how ordering could be naturally maintained. A further extension to this would be a queue-aware scheduler that natively supports the producer-consumer model. This is what the hardware graphics pipeline effectively has, and thus it would be possible to build a software-based feed-forward pipeline instead of a sort-middle pipeline. Numerous open questions remain related to configurability and design of such scheduling unit.

Finally, exposing the current hardware ROP in compute mode would provide a remarkable boost of up to 80%, depending on the MSAA mode, to the overall rasterization performance with fairly little hardware modifications required. Naturally, the freedom of programmability would be lost when using the hardware ROP, but there is no reason to neglect it when the blend function is one of the hardware-supported ones.

6.2 Future Work

Our pipeline can be used as a basis for numerous techniques that aim at reducing the shading workload. Among the simplest is the compaction of shading requests after a discard phase, and avoiding unnecessary quad expansion of pixels, as discussed above. Explicit cull programs [Hasselgren and Akenine-Möller 2007] could be used for the former purpose. More sophisticated quad merging [Fatahalian et al. 2010] allows the use of derivatives but avoids unnecessary work by merging neighboring pixels in quads even when they do not originate from the same triangle. In stochastic rasterization, shading can become very expensive unless decoupled sampling [Ragan-Kelley et al. 2011] based on shading caches are used. This is not possible to implement in the hardware graphics pipeline, but could be experimented with in the programmable pipeline. Unrestricted frame buffer access enables various multi-

fragment effects as discussed by Liu et al. [2010]. Finally, being able to modify the pipeline structure itself enables many exciting rendering paradigms, such as the combination of rasterization and ray tracing, as explored by Sugerman et al. [2009].

Our current implementation requires a separate rendering pass for each rendering state, but some workloads may require a more efficient approach. Stateless rendering, where each input triangle carries a state ID, is an attractive option and warrants further experiments. Our choice of buffering intermediate data between stages in large DRAM buffers, instead of locally executing different stages on-demand, is an efficient but somewhat crude solution because of having to resort to restarts in overflow situations. Dynamic scheduling of stages would alleviate this problem, and it may also be necessary in practice for pipelines that involve loops (e.g., ray tracing). This provides a challenging avenue for future work.

Adapting the bin and tile sizes to viewport dimensions would most likely be beneficial, but we have not experimented with this so far. Optimizing these constants for particular kinds of content could provide major speedups, especially when small viewports are used. Furthermore, it is a concern that some workloads may place exceptionally high burden on a few bins, causing low GPU utilization in the coarse rasterizer stage. We experimented with a scheme that allows two coarse rasterizer CTAs to work on the same bin when this happens, but found that the overall performance degraded except in highly specialized test cases. Nonetheless, more sophisticated load balancing schemes would deserve further research.

Our pipeline is roughly as fast as the hardware in processing culled triangles, but it has a relatively high constant per-triangle cost in other cases. This is because every non-culled triangle has to go through the bin and coarse rasterizer stages even if it ends up covering only one pixel in the end. It would be possible to detect very small triangles in the setup stage and implement a separate fast path that bypasses the two stages for such triangles. However, maintaining the correct rendering order is not trivial. In cases where the ordering requirements are not necessary, such as depth-only rendering, bypassing the stages could be a viable option for improving the performance with small triangles.

So far we have adamantly respected the rasterization order, but in some cases less strict ordering could be adequate. Whenever the user guarantees that a rendering batch contains no intersecting geometry and no order-dependent blending mode is active, we could rely on depth buffering producing the correct image. This kind of ordering where each batch is internally unordered is not supported by the current APIs or hardware, and it could enable new opportunities for optimization. Investigating how the pipeline could best exploit the relaxed ordering requirements, and quantifying how much speedup it offers, is an interesting future task.

7 Conclusions

We have presented a complete software rasterization pipeline on a modern GPU. Our results show that the performance of a thoroughly optimized, purely software-based pipeline is in many cases of the same magnitude as the performance of the hardware graphics pipeline on the same chip. Unlike the hardware pipeline, a software pipeline can be specialized to suite particular rendering tasks. This can involve both simplifications to gain better performance, and extensions to enable algorithms that the hardware graphics pipeline cannot accommodate. Due to its performance, full programmability, and strict adherence to central constraints imposed by the graphics APIs, our pipeline is a natural springboard for further research of programmable graphics on GPUs.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics 2009*, 145–149.

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic rasterization using time-continuous triangles. In *Proc. Graphics Hardware*, 7–16.

FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. 2010. Reducing shading on GPUs using quad-fragment merging. *ACM Trans. Graph. 29*, 67:1–67:8.

GASCUEL, J.-D., HOLZSCHUCH, N., FOURNIER, G., AND PEROCHE, B. 2008. Fast non-linear projections using graphics hardware. In *Proc. I3D*, 107–114.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical z-buffer visibility. In *Proc. SIGGRAPH '93*, 231–238.

HASSELGREN, J., AND AKENINE-MÖLLER, T. 2007. PCU: The programmable culling unit. *ACM Trans. Graph. 26*, 92:1–92:10.

LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. Nvidia Tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 39–55.

LIU, F., HUANG, M.-C., LIU, X.-H., AND WU, E.-H. 2010. Freepipe: A programmable parallel rendering architecture for efficient multi-fragment effects. In *Proc. I3D*, 75–82.

LOOP, C., AND EISENACHER, C., 2009. Real-time patch-based sort-middle rendering on massively parallel hardware. Microsoft Research tech. rep., MSR-TR-2009-83.

MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. 1994. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl. 14*, 23–32.

NVIDIA, 2001. HRAA: High-resolution antialiasing through multisampling. Tech. rep.

NVIDIA, 2007. Cuda technology; http://www.nvidia.com/cuda.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *ACM Trans. Graph. 21*, 3, 703–712.

RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. 2011. Decoupled sampling for graphics pipelines. *ACM Trans. Graph. 30*, 3, 17:1–17:17.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph. 27*, 18:1–18:15.

SUGERMAN, J., FATAHALIAN, K., BOULOS, S., AKELEY, K., AND HANRAHAN, P. 2009. Gramps: A programming model for graphics pipelines. *ACM Trans. Graph. 28*, 4:1–4:11.