Analyzing a Software Computer Graphics Pipeline: Implementation, Comparative Benchmarking and Pipeline Customization

Computer Science Senior Seminar Conference
Saturday, April 18th, 2020

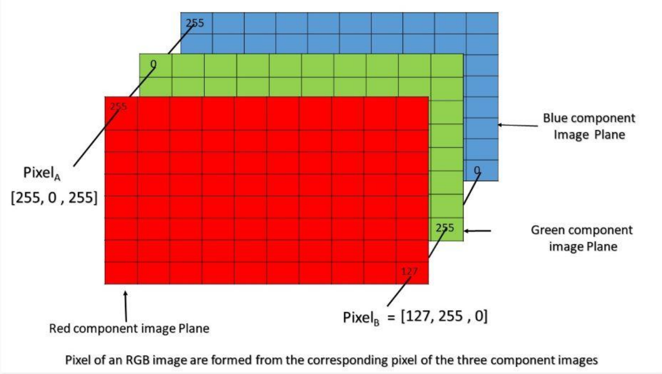
Kyle T Fluto
Division of Science and Mathematics
University of Minnesota - Morris
Morris, MN 56276
fluto006@umn.edu

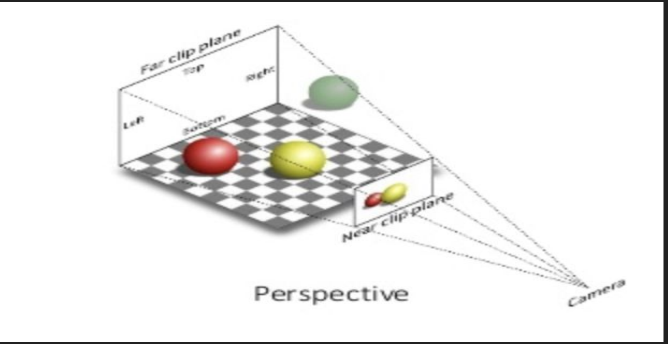
Talk Outline

- Introduction
- Background
  - Computer Graphics Fundamentals
- Pipeline Implementation: CUDA Rendering Engine (cuRE) (Deep Dive)
  - Streaming/Bounded in Memory
  - Vertex Reuse w/ Static Batching
- Comparative Benchmarking (Hardware vs. Software Performance)
- Novel Software Pipeline Extensions
  - Adaptive Subsampling
- Conclusion
- References & Acknowledgements

Background: Computer Graphics Fundamentals

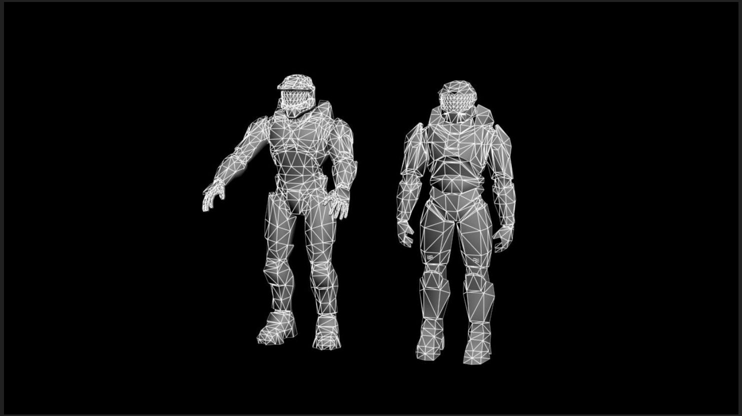
- Frames
- RGB Pixels
- Viewing Frustum
- Shaders → programs written for GPU (all pipeline stages...does not have to "shade")

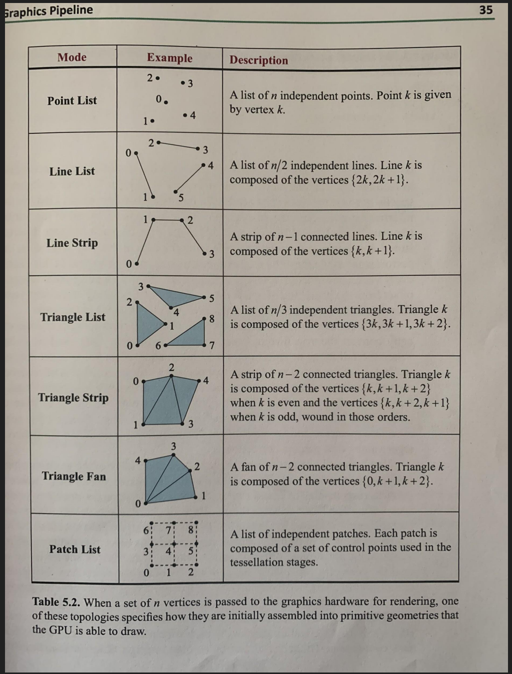




Background: Primitives and Mesh

- 3 raw primitives (points, lines, triangles)
- Various assembled primitives
- Triangle mesh





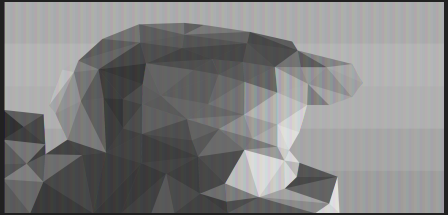
Graphics Pipeline 35

Mode	Example	Description
Point List		A list of n independent points. Point k is given by vertex k.
Line List		A list of $n/2$ independent lines. Line k is composed of the vertices $\{2k, 2k+1\}$.
Line Strip		A strip of $n-1$ connected lines. Line k is composed of the vertices $\{k, k+1\}$.
Triangle List		A list of $n/3$ independent triangles. Triangle k is composed of the vertices $\{3k, 3k+1, 3k+2\}$.
Triangle Strip		A strip of $n-2$ connected triangles. Triangle k is composed of the vertices $\{k, k+1, k+2\}$ when k is even and the vertices $\{k, k+2, k+1\}$ when k is odd, wound in those orders.
Triangle Fan		A fan of $n-2$ connected triangles. Triangle k is composed of the vertices $\{0, k+1, k+2\}$.
Patch List		A list of independent patches. Each patch is composed of a set of control points used in the tessellation stages.

Table 5.2. When a set of n vertices is passed to the graphics hardware for rendering, one of these topologies specifies how they are initially assembled into primitive geometries that the GPU is able to draw.

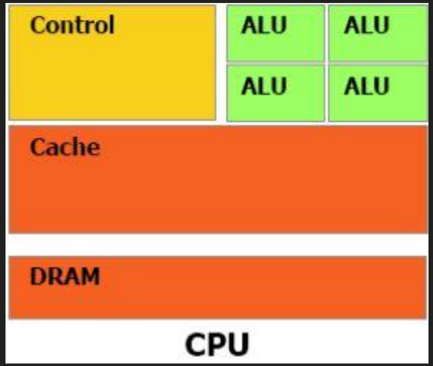
Background: Computer Graphics Fundamentals

- As resolutions and scene fidelity increase, more polygons are needed
- This is costly → modern scenes have pixels in the millions and need to run at 30-60 fps in order to provide a useable experience
- Hardware and software optimizations are needed to maintain high throughput through the pipeline

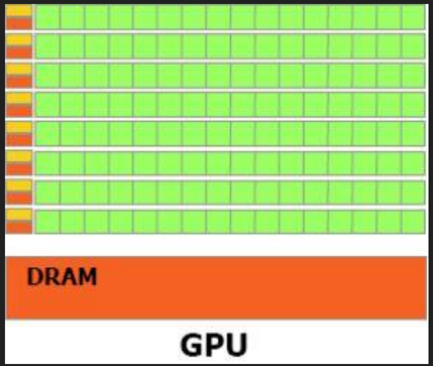


Background: GPU Hardware Architecture

- CPU Architecture
  - Serial Execution
  - Limited threading capability

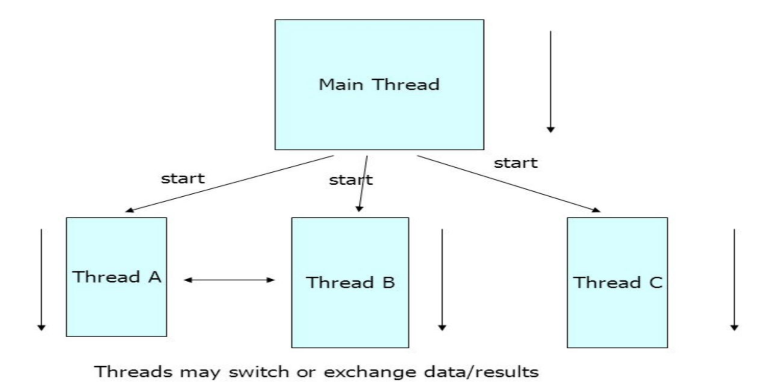


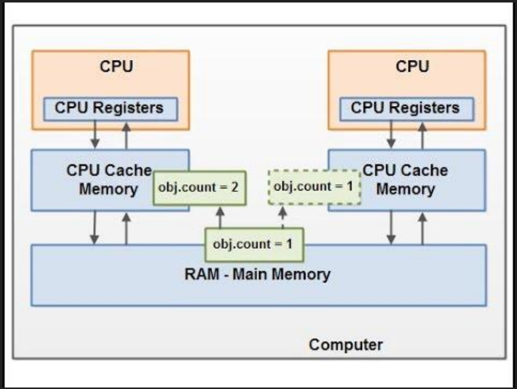
- GPU Architecture
  - Streamed processing (linear)
  - Cached data between pipeline stages
  - Parallel Execution (multiple streams)



Background: Threads & Parallelism / Registers & Cache

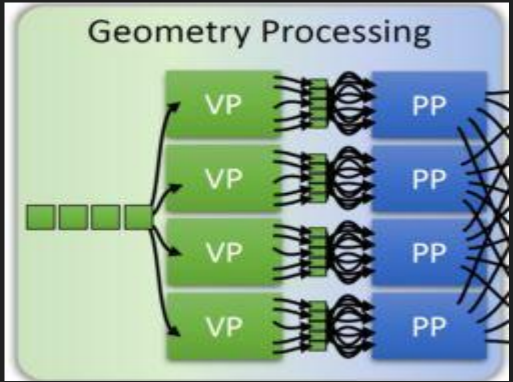
- Threads are subprocesses of a parent process and allow subproblems to be solved in parallel
- Threads operate locally on the same processor (Batches & Warps) for dedicated portions of the screen
- Registers are the fastest on-chip memory followed by cache. Locality == lower latency

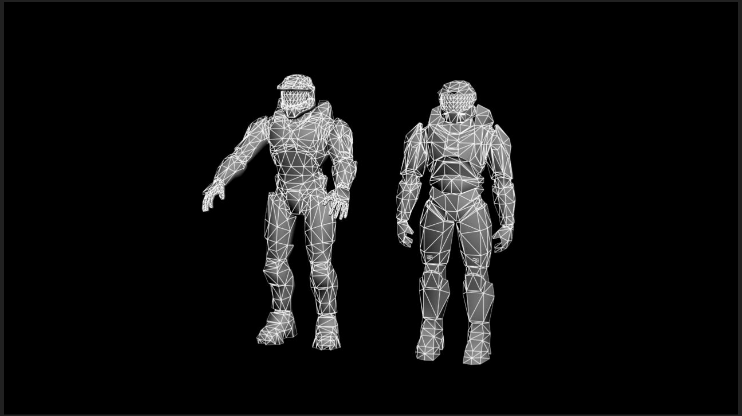




Background: Pipeline Stages (Geometry Processing)

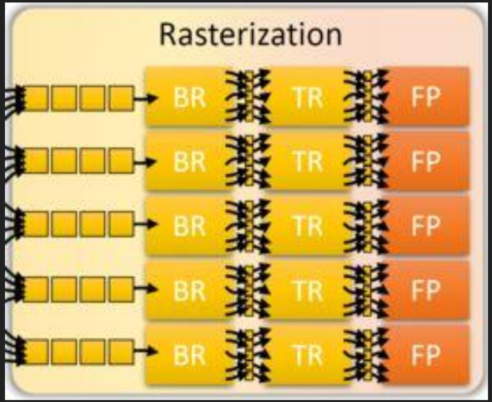
- Responsible for generating the triangle mesh from primitives
- OpenGL/Graphics Utility → feeds raw primitives (points, lines, triangles)
- Vertex processing is used to correctly position the primitives inside of the viewing frustum
- Primitive processing assembles the transformed vertices into the triangle mesh and into “clip-space”

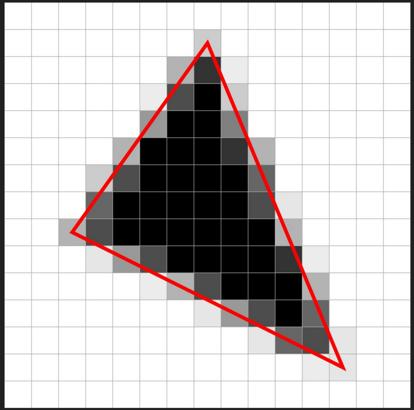




Background: Pipeline Stages (Rasterization)

- Responsible for shading the pixels that are included in the scene geometry
- Clipping/Culling Tests → (outside/overlapping)
- Determines pixel coverage with full/partial screen pass
- Uses interpolation to calculate pixel values contained within triangles
- Lighting and coloring

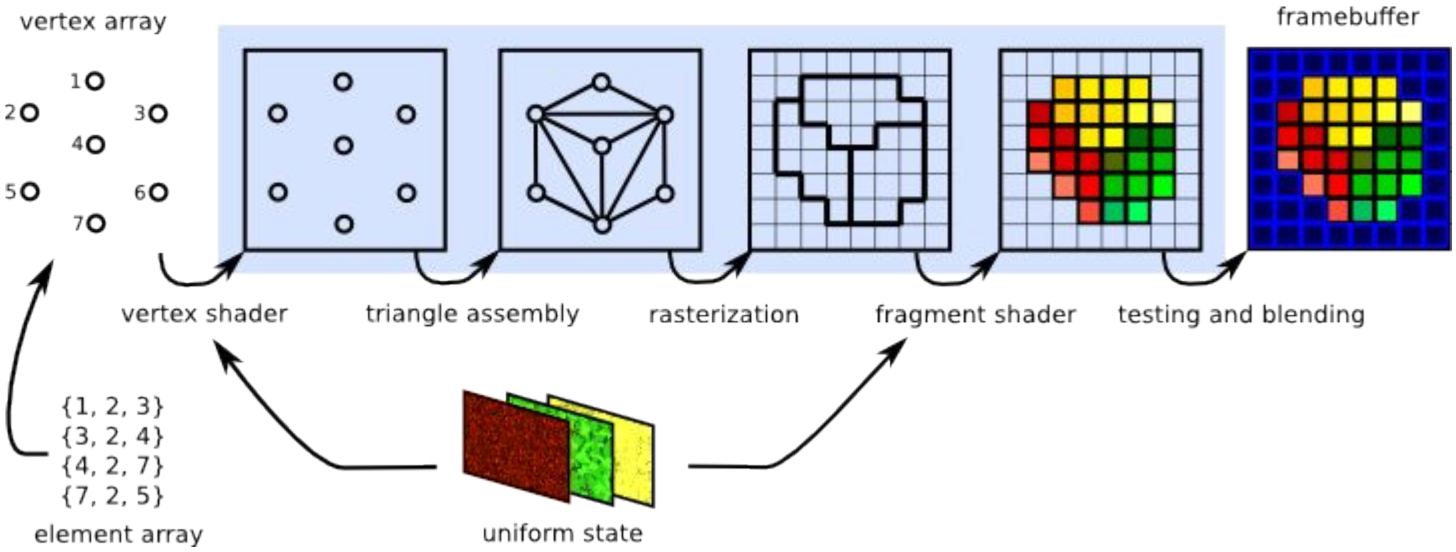




Background: Pipeline Stages (Frame Buffer Operations)

- Primary responsibilities
    - Stores the final pixel data for each frame
    - Heuristic tests (depth tests)
    - Color blending

Full Pipeline

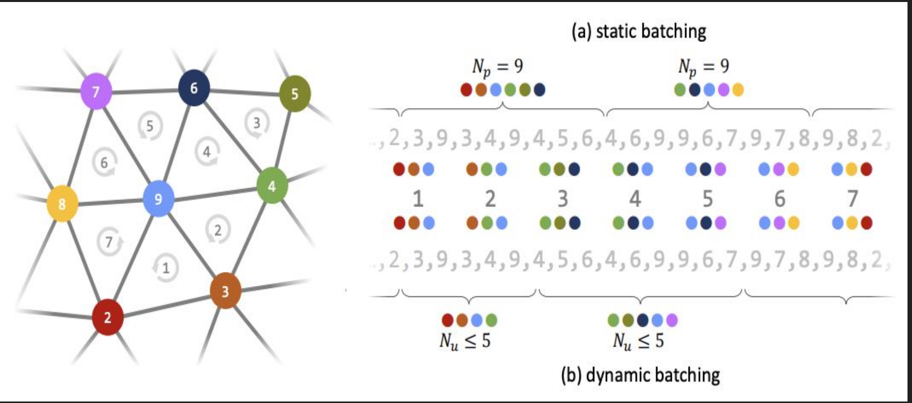


Pipeline Implementation - CUDA Rendering Engine (cuRE)

- Design Considerations/Goals:
  - Create a pipeline entirely in software ⟶ create programmable versions of the fixed function stages of the pipeline (primitive processing and rasterization)
  - Streaming -> parallel architecture
  - Bounded in memory (no global RAM use, just local GPU resources) ⟶ vertex reuse w/ static batching and warp-level register shuffling
  - Custom shaders

Pipeline Implementation - Vertex Reuse with Static Batching and Warp Level Register Shuffles

- Geometry processing → vertex shader once for each primitive in index buffer
- Same vertex is referenced 6 times in a typical scene
- Wasted use of post-transform cache
- Redundant computation



Pipeline Implementation - Vertex Reuse with Static Batching and Warp Level Register Shuffles

- But, repeated vertices are local in proximity
- Introduce warp level register shuffles for intra-thread communication
- Threads publish their assigned vertex and compare with other threads in the warp

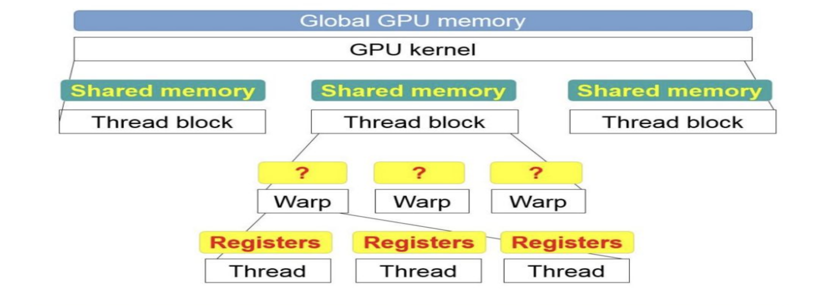


Figure 1: Execution and Memory hierarchy in CUDA GPUs.

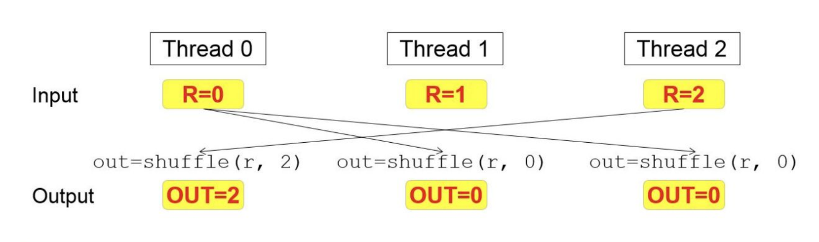


Figure 2: Shuffle semantics.

Pipeline Implementation - Vertex Reuse w/ Static Batching

- Static Batching ⟶ fixed portion of index buffer is assigned to synchronized threads within a warp (ideally all unique)
- Each thread compares itself within the warp by performing a register shuffle (bitmask XOR of current and target thread register values)
- If bitmask is 0 after shuffling, the value is not stored in the vertex map/register
- Vertex map determines assigns vertices to triangles

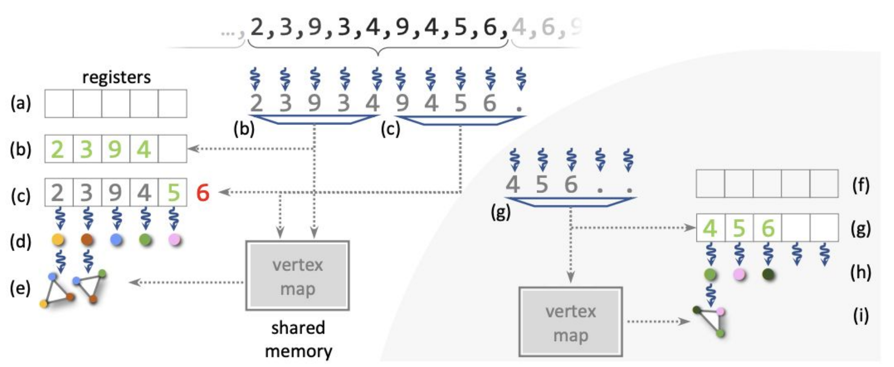


Fig. 3. Statically-batched warp voting uses all threads in a warp (5 in this example) to load indices. (b) We exploit warp voting and shuffle instructions to unify the indices and store the result in registers. (c) This process is repeated until all indices have been consumed, or all threads have acquired 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).

shflsync(m,r,t) -> m = 32-bit bitmask of threads in warp, r = register, t = target thread

Comparative Benchmarking: Vertex Shader

- Average Shading Rate (ASR) → vertex shader invocations ÷ triangles in scene
- Lower rate is better
- cuRE w/static batching vertex reuse performs very close to OpenGL running on native hardware

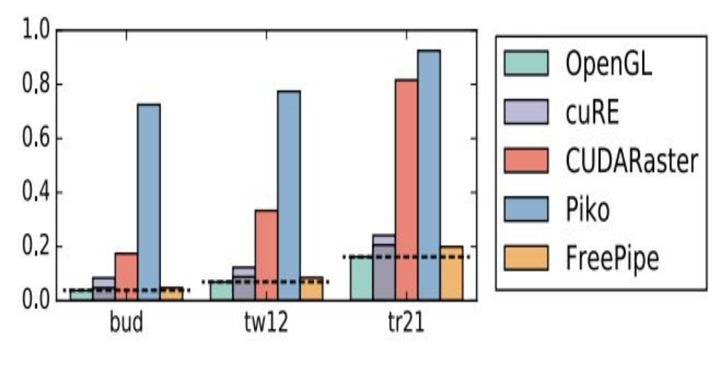
				Parallel Cache			OpenGL			Ours	
	vert	tris	ideal	1024	2048	4096	NVIDIA	AMD	Intel	stat.w	dyn.*
bunny	34k	70k	0.504	2.832	2.820	2.799	0.879	0.770	0.571	0.858	0.603
sphere	40k	82k	0.501	2.811	2.805	2.793	0.884	0.772	0.584	0.864	0.615
tree	492k	239k	2.058	2.997	2.997	2.997	2.078	2.061	2.059	2.058	2.061
buddha	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

Comparative Benchmarking: Frame Draw Times & Memory Use

Frame draw times are in ms

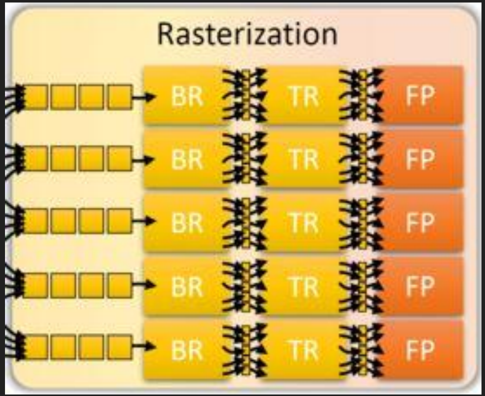
- cuRE ran against 3 other existing software pipelines (Piko, FreePipe, CUDARaster) and native hardware via OpenGL
- Over 100 frames from current video games were used (e.g. Deus-ex, Total War, etc.)
- cuRE ran within 1 order of magnitude (power of 10) of hardware w/ OpenGL
- cuRE was economic with GPU memory use

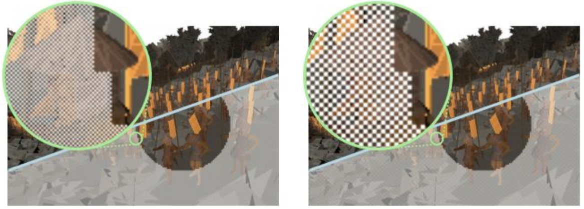
		bud	fry	san	sib	tr12	tw12	sh23	sg13
GTX 1080	OpenGL	1.1	0.2	7.5	0.1	2.9	2.3	0.8	0.4
	OpenGL_{fi}	1.2	0.3	7.5	0.1	4.0	3.9	2.1	0.9
	CUDARaster								
	cuRE	7.8	2.8	35.8	1.6	28.4	39.9	16.8	6.4
	$\text{cuRE}_{w/q}$	5.4	2.4	29.2	1.4	24.0	21.5	14.8	6.1
	$\text{cuRE}_{w/p}$	8.3	2.8	35.5	1.4	25.3	37.9	14.2	5.8
	$\text{cuRE}_{w/o}$	5.5	2.3	28.0	1.4	20.9	19.4	12.1	5.4
	Piko	8.4	3.6	44.0	2.9	37.1	25.1	12.5	7.7
	FreePipe	0.8	72.3	292.7	68.1	261.3	66.2	141.4	145.0
GTX 780 Ti	OpenGL	1.8	0.4	12.3	0.2	5.1	3.4	1.4	0.8
	OpenGL_{fi}								
	CUDARaster	4.2	2.1		1.5	20.1	14.3	7.1	4.6
	cuRE	23.1	8.0	143.3	4.8	88.9	95.2	37.4	18.3
	$\text{cuRE}_{w/q}$	17.1	7.4	105.7	4.1	70.7	56.9	33.8	16.7
	$\text{cuRE}_{w/p}$	23.6	7.8	143.7	4.3	82.8	93.5	31.1	16.7
	$\text{cuRE}_{w/o}$	16.9	7.0	95.6	3.6	63.5	50.4	28.3	16.7
	Piko	19.4	8.5		7.5	89.9	62.3	28.9	19.8
	FreePipe	2.1	156.3		149.1	903.6	182.5	463.8	492.4



Novel Pipeline Extension: Adaptive Subsampling

- Subsampling → reduce the # of pixels rendered in parts of the scene outside of the focal point of the scene. Very important in VR display latency
- Removes thread divergence (same computation) → coverage shader performs logical AND (with coverage mask built during the Tile Rasterizer (TR) → Less pixels





(a) 1 × 1 pixel pattern

(b) 2 × 2 pixel pattern

Fig. 14. Using our programmable coverage shader stage, we can implement adaptive checkerboard rendering without the inefficiencies of the conventional approach based on discarding fragments. The images above show a scene captured from the game *Total War: Shogun 2* rendered with adaptive checkerboard rendering using (a) 1 × 1 and (b) 2 × 2 pixel squares.

Conclusion

- Hardware is fastest
- The cuRE rendering engine comes within an order of magnitude (power of 10) of hardware via OpenGL graphics utility
- cuRE can create custom shaders that do not exist in native hardware for special applications

References & Acknowledgements

References

1. M. Kenzel, B. Kerbl, D. Schmalstieg, and M. Steinberger. A high-performance software graphics pipeline architecture for the gpu. ACM Trans. Graph., 37(4), July 2018.
2. M. Kenzel, B. Kerbl, W. Tatzgern, E. Ivanchenko, D. Schmalstieg, and M. Steinberger. On-the-fly vertex reuse for massively-parallel software geometry processing. Proc. ACM Comput. Graph. Interact. Tech., 1(2), Aug. 2018.
3. Hamilis, Matan and Silberstein, Mark. Register cache: Caching for warp-centric cuda programs, 2017. [Online; accessed 11-March-2020].
4. Lin, Yuan and Grover, Vinod. Using cuda warp-level primitives, 2018. [Online; accessed 11-March-2020].

Acknowledgements

1. Professor Peter Dolan - Senior Seminar Advisor
2. Professor Elena Machkasova - Academic Advisor
3. Professor Kristin Lamberty - CSci 4901 Course Instructor
4. Trent Merkins - UMM CSci Student (practice talk feedback)

This was an extremely humbling experience and undertaking. Respect to individuals that conduct research/contribute to the body of science in their chosen fields. Thanks for the great times in virtual worlds.