Active Thread Compaction
for
GPU Path Tracing

Ingo Wald
Ingo.Wald@intel.com

High-Performance Graphics 2011, Vancouver, CA

Outline

Active Thread Compaction

- The concept
- A “proof of concept” implementation (for Path Tracing)
- Experimental Results

Analysis

- Or: “Why the heck does it *still* not getting faster?”

Motivation

Modern archs increasingly rely on SIMD for perf

- CUDA/Fermi: 32 “threads” in a SIMD “warp”

Compilers/languages help avoid “ugliness” of SIMD…

- OpenCL, CUDA, shading languages … give appearance of “scalar” code

… but HW is *still* SIMD

- If one *thread* wants to do an op, entire *warp* does it
- Code divergence → HW execute both paths, each w/ predication
- Lots of divergence: potentially running at utilization of 1/32nd

SIMD divergence very much real

Ray Traversal

- Currently traversed node: inner node vs leaf node
  - … and leaves can have vastly different lengths
  - And what if we support more than just triangles?
- Varying number of traversal steps until completion
- …

Path tracing itself

- Traversal found intersection or not
- Different material type, absorption or bounce
- Valid light sample?
- In Shadow?
- …

Active Thread Compaction

Active Thread Compaction

Basic Idea: If cost/warp is indep of #active threads…

- (that's the basic assumption of SIMD)

… and if we have lots of low-utilized warps …

… then "merging" low-utilized warps should save work

- Eg, take 10 warps of 3 paths each; make 1 warp of 30 paths →10x win

Conceptually simple to implement

- Write all (logical) thread states into array
- Perform stream compaction
- Read "thread" state back (into other HW thread)

Active Thread Compaction

Important: Two ways to look at “compaction”:

- A “Library Tool” / programming paradigm
  - Think in “streams” of data
  - Call kernel on streams, perform compaction of streams, etc.

Active Thread Compaction

Important: Two ways to look at “compaction”:

- A “Library Tool” / programming paradigm
  - Think in “streams” of data
  - Call kernel on streams, perform compaction of streams, etc.
- A feature of the programming language
  - Think in “blocks” and “warps” of “threads”
  - Compaction is “language primitive” that (re)groups threads into warps
    - Like __syncThreads, but w/ change of thread→warp mapping

    →“__compactThreads()” ???
    - But: not the focus of this paper….

ATC for GPU Path Tracing

Focus/Motivation

- **NOT** primarily “making path tracing fast/nice on GPUs”
- Rather: Test case to evaluate ATC *concept*
  - *Ideal* test-case: Lots of divergence, huge potential for savings…

Experimental Infrastructure

CUDA ray tracing core

- Binary BVH w/ SAH, but no spatial splits
- Aila-"like" traversal, but simplified (eg, no use of texture mem)
- Important: call traversal core directly from within path tracer
  - Ie, *not* as stand-along kernel operating on set of nicely sorted array of rays
- Can't match the numbers in the Aila-paper, but "reasonably" fast…

Experimental Infrastructure

- CUDA ray tracing core
- Intentionally simple CUDA Path Tracer Infrastructure
  - Padded replication sampling (Scrambled Hammersley pattern)
  - “Uber”-Material that switches between several Material types
  - Simple path tracer: Generate initial path, then iterate:
    - Trace path to next hitpoint, terminate if none found.
    - Sample light source, terminate if invalid sample
    - Shoot shadow ray, evaluate BRDF and accumulate “light” if illuminate
    - Compute outgoing ray by sampling BRDF; terminate if absorbed (RR)

Experimental Infrastructure

- CUDA ray tracing core
- Intentionally simple CUDA Path Tracer Infrastructure
- Three different kernels
  - "Naïve", "whole-frame", and "tiled"

1) Naïve Reference Kernel

Reference kernel: Naïve CUDA implementation

- Each pixel is one path; each path is one CUDA thread
- Each thread does whole traversal loop until terminated

Pro:

- That's how you *want* to write a path tracer

Con:

- SIMD divergence

2) “Whole Frame Compaction”

Not actually a “kernel”, rather entire framework

- CUDA kernel itself does only one “bounce” (trace→illuminate→sample)
  - Reads path node from dedicated array, performs bounce, writes back result (marks terminated paths in separate, dedicated “pathActive” array)
- Outer “for all bounces” loop done by app
- Compaction: Host calls compaction primitive in-between two bounces
  - Use CUDPP library [Harris et al]

2) “Whole Frame Compaction”

Pro

- In theory, should make (most) warps fully occupied
- Still divergence *inside* bounce kernel, but fully occupied at each call

Con

- That’s *not* how you want to code
- Lots of possible sources of overhead
  - Multiple memory read/writes per node; coherence/locality; …
  - Explicit dependencies between host- and device-codes

3) “Tile-based Compaction”

Compaction, but only inside each screen tile (→block)

- Still iterate “read→bounce→write” …
- … but all control flow on device (outer loop in same kernel)
- All arrays are local arrays, no need for app to alloc arrays, call kernels,…

Pro:

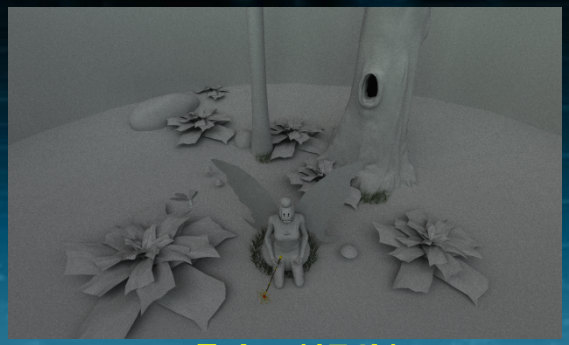
- Much nicer programming model
  - In particular: Could do that transparently from within a compiler
- Better locality, no host-device dependencies
- Can use shared rather than global memory for reordering(!)

Experimental results

Experimental results

Various artificial test scenes (170k – 2.6m tris)

- All “open”, all “diffuse only”, 8 bounces, russian-roulette@5%
- Screen res 1280x728, 1 path per pixel per frame (w/ accumulation)

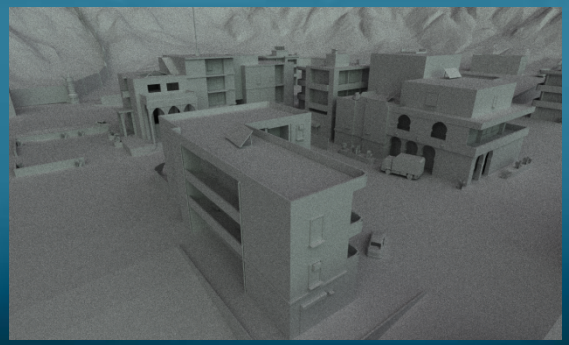


Fairy (174k)

DreamHome (2.63 M)

DragonTemple (922k)

TrollTemple (622k)



Refinery (297k)

Moto (519k)

Experimental results

Various artificial test scenes (170k – 2.6m tris)

- All “open”, all “diffuse only”, 8 bounces, russian-roulette@5%
- Screen res 1280x728, 1 path per pixel per frame (w/ accumulation)

Hardware: GTX480, hand-tuned parameters

- Registers per thread = 64 (everything else is worse)
- Num threads per block = 64 (8x8 for naïve, 64x1 for others)

Experimental results

Speedups: Generally "underwhelming"

- At best 12-16 *percent* speedup (rather than 3x!)
- Tiled/shared mem actually 3x *slower*

scene	naïve	whole-frame comp.		tiled (shared mem)	
Path Tracing, max. path length=8					
fairy	3.13	3.54	+13%	1.08	-66%
moto	2.03	2.29	+13%	0.74	-64%
troll	3.19	3.63	+14%	1.10	-66%
dragon	3.61	4.06	+12%	1.21	-66%
dreamhome	3.42	3.88	+13%	1.23	-64%
refinery	4.17	4.84	+16%	1.42	-66%

Now *why* is it so slow?

"Maybe" it's a bug? Or flawed assumptions?

Now *why* is it so slow?

In part: ***3x slower*** for tiled SHM kernel?

– 3x slower for kernel we thought *best* ???

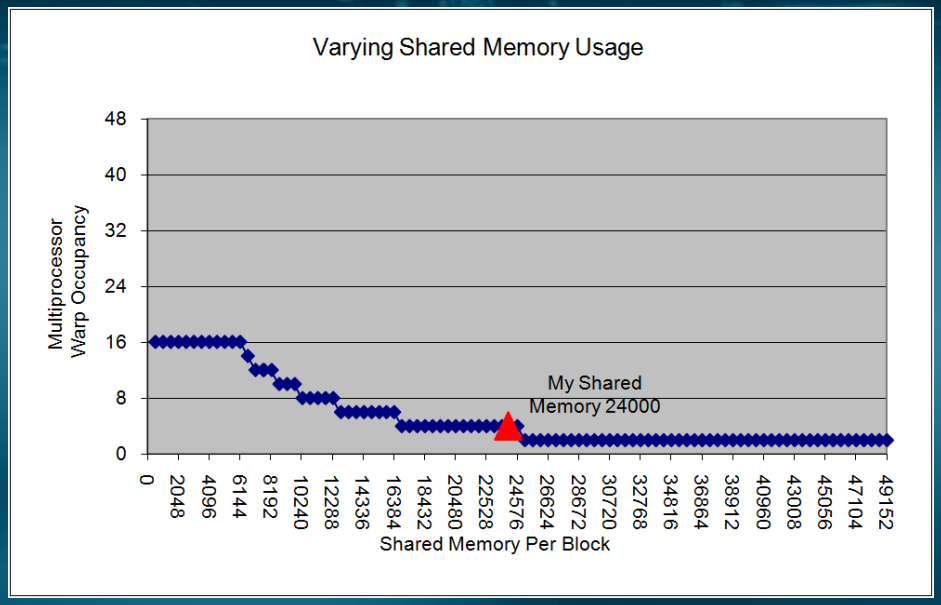
Now *why* is it so slow?

In part: ***3x slower*** for tiled SHM kernel?

- 3x slower for kernel we thought *best* ???

Explanation: Look at "CUDA Occupancy Calculator"

- In particular, plot "occupancy over SHM usage"
  - Constraint: 64 registers per thread (best perf. config)



→*At best* 16% occupancy (32% dev util)

→@24K/thread block: ~4% occupancy (8% util)

→Using ~24KB SHM makes us lose 4x in device utilization! (can't make that up w/ compaction)

CUDA Device Occupancy

Interesting: SHM-util. issue NOT specific to our app

Occupancy explains the 3x slowdown for SHM kernel

- 4x loss in device utilization is hard to make up for

But why is the (non-SHM) whole-frame kernel so slow?

- Bug? Flawed assumption?

Now *why* is it so slow?

"Maybe" it's a bug? Or flawed assumptions?

– To find out, look at execution statistics:

	active	naive		compact	
	paths	#warps	paths/warp	#warps	saved
0	983 (**983**)	30,720 (**31k**)	32 (**32**)	31k (**31k**)	**1**×
1	983 (**1,966**)	30,720 (**61k**)	32 (**32**)	31k (**61k**)	**1**×
2	528 (**2,494**)	30,717 (**92k**)	17 (**27**)	17k (**38k**)	**1.2**×
3	352 (**2,846**)	30,698 (**123k**)	11 (**23**)	11k (**89k**)	**1.4**×
4	225 (**3,071**)	30,144 (**153k**)	7.5 (**20**)	7k (**96k**)	**1.6**×
5	155 (**3,226**)	28,655 (**182k**)	5.4 (**17.8**)	4.8k (**101k**)	**1.8**×
6	108 (**3,334**)	26,383 (**208k**)	4.1 (**16**)	3.4k (**104k**)	**2**×
7	80 (**3,414**)	23,450 (**231k**)	3.4 (**14.7**)	2.5k (**107k**)	**2.2**×
8	60 (**3,474**)	16,651 (**250k**)	3.2 (**13.8**)	1.9k (**108k**)	**2.3**×

→In fact, fully in line with expectations… (2.3x fewer warp-bounces)

Now *why* is it so slow?

So it works in theory… but why not in practice !?

Overhead for doing compaction?

- No: checked cost(compaction), is ~ 1%

Now *why* is it so slow?

So it works in theory… but why not in practice !?

Overhead for doing compaction?

- No: checked cost(compaction), is ~ 1%

Overhead for storing/loading paths?

- No: checked naïve vs whole-frame w/o compaction

Now *why* is it so slow?

So it works in theory… but why not in practice !?

Overhead for doing compaction?

- No: checked cost(compaction), is ~ 1%

Overhead for storing/loading paths?

- No: checked naïve vs whole-frame w/o compaction

Too few rays available for compaction to work?

- No – ran both 1920x1280 and 1280x768; no difference at all

Time per warp-bounce

"warp bounce": one warp doing one "bounce" kernel

(trace→illuminate→shadow→sampleBRDF)

Look at "time/warp-bounce", wrt path depth:

	0	1	2	3	4	5	6	7	8
compaction disabled									
paths/warp	32	32	17	11	7.5	5.4	4.1	3.4	3.2
time/warp	1.6	2.3	1.9	1.4	1.1	1.0	0.9	0.8	0.9
compaction enabled									
paths/warp	32	32	32	32	32	32	32	32	32
time/warp	1.6	2.3	3.1	3.2	3.4	3.6	3.8	4	4.3

→No compaction: Low-utilized warps get *faster*

→W/ compaction: Time/warpbounce *increases* significantly

→Improving utilization *increases* cost per kernel call (@same #insts)

Time per warp-bounce

Full warps more exp. than 1-thread warps?

- (in theory, they shouldn't be – at least, not that bad)

Explanation 1: Code divergence

- More rays in warp → higher prob that rays' code path diverges
- More rays in warp → higher prob that one of them is extra expensive

- Both true, but neither fully conclusive…

Time per warp-bounce

Explanation 2: Memory

- Fact: we have barely enough threads to occupy device at all…
  - 64 threads/block → *two* warps per block
  - Certainly not enough to hide O(1000's) cycles in latency
- Device has to serialize incoherent reads → latencies add up
  - read of *32* incoherent addresses *much* more costly than 1 or 2.

  → Increasing #threads/warp *increases* cost/warp

In practice, probably a combination of '1' *and* '2'

- SIMD divergence *and* (incoherent) memory accesses

Summary

Summary

Proposed concept of “active thread compaction”

- *Eventual* vision requires language/compiler support

Summary

- Proposed concept of “active thread compaction”
  - *Eventual* vision requires language/compiler support
- Shown significant statistical wins (at least for PT)
  - Reduction in core kernel calls: more than 2x!
  - But, grain of salt: avg is “only” 2.3x, not “10x”

Summary

- Proposed concept of “active thread compaction”
  - *Eventual* vision requires language/compiler support
- Shown significant statistical wins (at least for PT)
  - Reduction in core kernel calls: more than 2x!
  - But, grain of salt: avg is “only” 2.3x, not “10x”
- Shown that on today’s HW it doesn’t yet work well
  - “Complex” kernels have too few threads running to hide latencies
  - Low occupancy when using local store
    - (Re-)investigate for other/next-gen HW archs (?)

Summary

- Proposed concept of “active thread compaction”
  - *Eventual* vision requires language/compiler support
- Shown significant statistical wins (at least for PT)
  - Reduction in core kernel calls: more than 2x!
  - But, grain of salt: avg is “only” 2.3x, not “10x”
- Shown that on today’s HW it doesn’t yet work well
  - “Complex” kernels have too few threads running to hide latencies
  - Low occupancy when using local store
    - (Re-)investigate for other/next-gen HW archs (?)
- Today’s HW not at all sufficiently understood
  - In part, ray traversal by far not as “coherence-oblivious” as thought
    - Even for “simple” settings (single mesh, triangles only, …)

Questions

backup…

ATC for GPU Path Tracing

Focus/Motivation

- **NOT** primarily “making path tracing fast/nice on GPUs”
- Rather: Test case to evaluate ATC *concept*
  - Well-understood application, well-understood building blocks (ray traversal)
  - Code you *want* to code in scalar form
  - Highly variable work/thread; random – and frequent – “dying” of threads

    → Huge potential for ACT to give tangible benefit.

Other factors

Switch to speculative traversal kernel

- Bigger *relative* speedup, but even more mem I/O→ lower *absolute* perf

Artificially more compute (shade, isec)

- Yes, get bigger speedup … but no “useful” application

Ambient Light vs HDRI Light

- No big difference: HDRI light far more expensive …
  … but also completely latency-bound

Impact of geometry type and material types

- General rule: The less incoherence, the lower the benefit of compaction

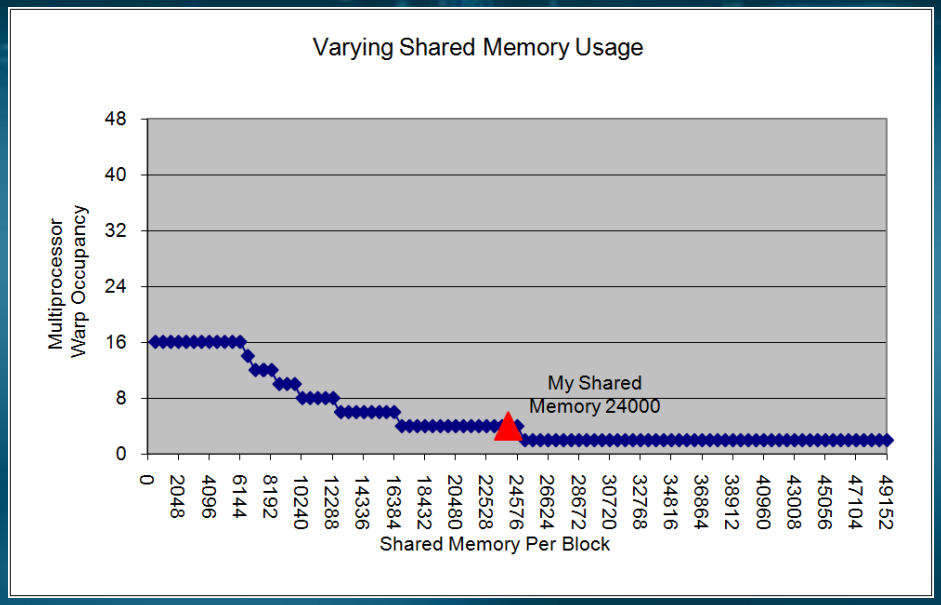
Now *why* is it so slow?

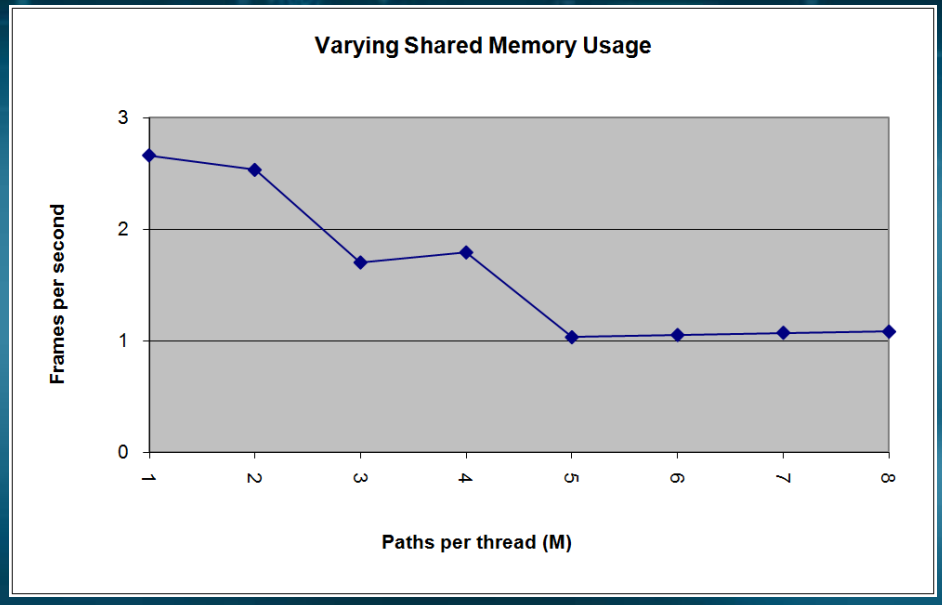
In part: ***3x slower*** for tiled SHM kernel?

- 3x slower for kernel we thought *best* ???

Explanation: Look at "CUDA Occupancy Calculator"

- In particular, plot "occupancy over SHM usage"
  - Constraint: 64 registers per thread (best perf. config)





Dealing with reduced utilization

Speculative execution (eg, [Aila et al])

"Re-fill" early-terminated threads [Novak et al]

Switch to other way of using SIMD

- "Horizontal" vs "vertical" SIMD [Kalojanov, Ernst, Wald, Waechter, …

Use some form of compaction-based traversal

- SIMD Stream tracing [Gribble,Wald,Tsakok]; packet reordering [Boulos]
- Stream compaction [von Antwerpen]

(this list is incomplete)

Active Thread Compaction

Example: Start w/ block of 4 warps, 4 threads each (all active)

→ 4 active warps, 16 useful ops

Active Thread Compaction

Example: Start w/ block of 4 warps, 4 threads each (all active)

→ 4 active warps, 16 useful ops

Bounce(path) /* 1st gen */

→ 4 active warps, only 9 useful ops (56%)

Active Thread Compaction

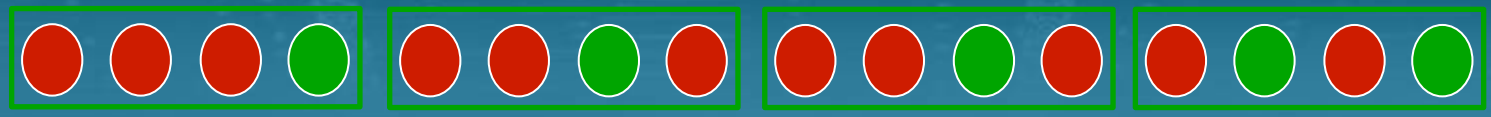
Example: Start w/ block of 4 warps, 4 threads each (all active)

→ 4 active warps, 16 useful ops

Bounce(path) /* 1st gen */

→ 4 active warps, only 9 useful ops (56%)

Buonce(path) /* 2nd gen */



→ 4 active warps, only 5 useful ops (31%)

Active Thread Compaction

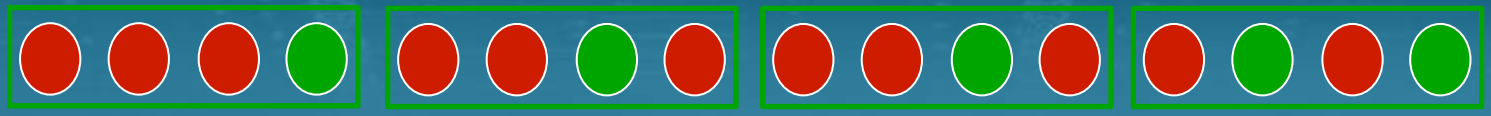
Example: Start w/ block of 4 warps, 4 threads each (all active)

→ 4 active warps, 16 useful ops

Bounce(path) /* 1st gen */

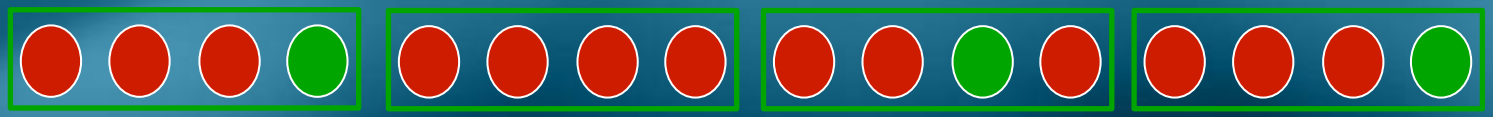
→ 4 active warps, only 9 useful ops (56%)

Buonce(path) /* 2nd gen */



→ 4 active warps, only 5 useful ops (31%)

…

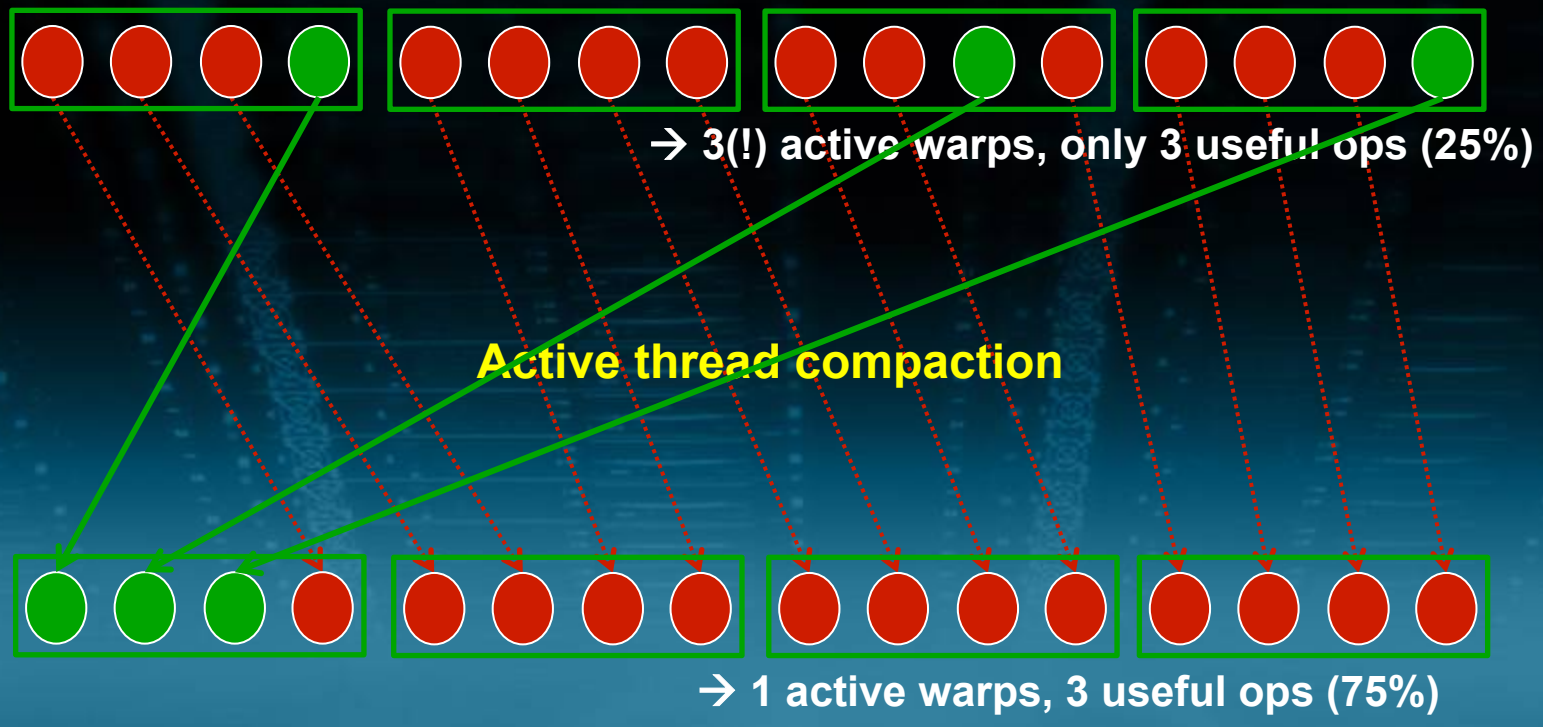


→ 3(!) active warps, only 3 useful ops (25%)

Active Thread Compaction

→ 3(!) active warps, only 3 useful ops (25%)

Active Thread Compaction



Same actual work done by 1 warp (rather than 3) → 3x win!

ATC for GPU Path Tracing

Stop here: Just *how* big *is* this potential?

- Assume 8 bounces
- Assume 50% chance of path "dying" (lost to env, RR-absorption)
  - To maximize this, pick "open" scenes.
- Then: After 5 bounces we're down to "1 out of 32" active threads

ATC for GPU Path Tracing

Stop here: Just *how* big *is* this potential?

- Assume 8 bounces
- Assume 50% chance of path "dying" (lost to env, RR-absorption)
  - To maximize this, pick "open" scenes.
- Then: After 5 bounces we're down to "1 out of 32" active threads

That's too naïve – in practice, win isn't all that big….

- 1st generation (primary rays) is 100% utilized → 32/32
- 2nd generation (1st bounce) is ~100% utilized → 32/32
  - Primary bounce points are coherent – all miss, or all hit
- 3rd generation is ~50% utilized (16/32)
- …
- Average across 9 generations (8 bounces) is ~10.6/32 → 3x benefit